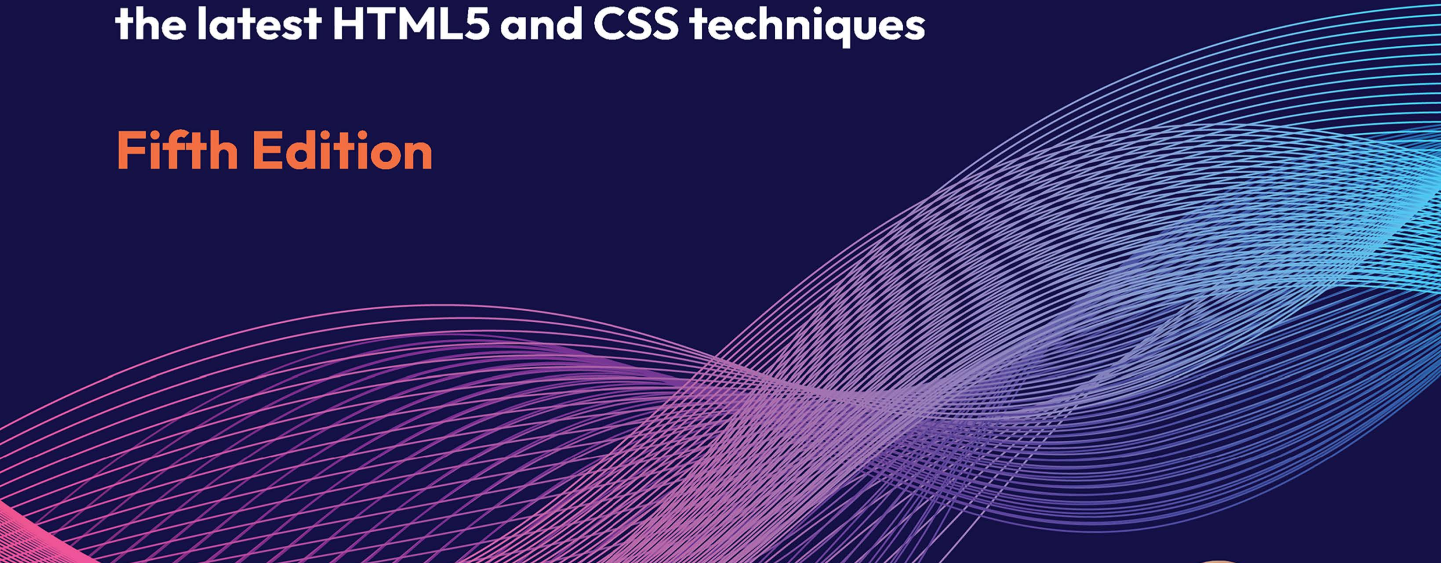
EXPERT INSIGHT

Responsive Web Design with HTML5 and CSS

Build future-proof responsive websites using the latest HTML5 and CSS techniques

Fifth Edition



Ben Frain

packt

Responsive Web Design with HTML5 and CSS

Fifth Edition

Build future-proof responsive websites using the latest HTML5 and CSS techniques

Ben Frain

<packt>

Responsive Web Design with HTML5 and CSS

Fifth Edition

Copyright © 2025 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Ashwin Nair
Relationship Lead: Aaron Lazar
Project Manager: Ruvika Rao
Content Engineer: Kinnari Chohan
Technical Editor: Rohit Singh

Copy Editor: Safis Editing

Indexer: Pratik Shirodkar
Proofreader: Kinnari Chohan
Production Designer: Ganesh Bhadwalkar
Growth Lead: Priyadarshini Sharma

Bonus Content authored by Michelle Manemann (Technical Reviewer): We are grateful to Michelle for contributing to the downloadable content that accompanies this book, ensuring readers have additional resources to deepen their understanding.

First published: April 2012
Second edition: August 2015
Third edition: April 2020
Fourth edition: September 2022
Fifth edition: October 2025

Production reference: 2151025

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-83702-823-8

`www.packtpub.com`

I dedicate this book to the memory of my dad – the best father and friend a boy could ever be lucky enough to have.

– Ben Frain

Contributors

About the author

Ben Frain has been a web designer/developer since 1996. He is currently employed as a lead product designer at Bet365. Before the web, he worked as an underrated (and modest) TV actor and technology journalist, having graduated from Salford University with a degree in media and performance. He has written four equally underrated (his opinion) screenplays and still harbors an, admittedly fading, belief he might sell one.

He likes to pretend he isn't a nerd but spends an inordinate amount of time building keyboards and talking about them, text editors, and the web on his blog and YouTube channel. Outside of geekery, he enjoys managing his youngest son's football team, if standing in a freezing wet field, while a team of 11-year-olds shout at you to shut up can be called enjoyment. He is also hoping that, one day soon, his eldest teenage son will emerge from his room to spend some time with him and his wife, rather than merely to ask whether there is anything to eat or to ask for money.

Being five editions in, I am no longer surprised at the length of time it takes to write a new version of this book. But there were additional unforeseen bumps along the road this time, and I appreciate the Packt team's understanding.

I'd also like to extend thanks to the reviewers, both editorially and technically. Extra thanks to Michelle for adding the additional appendices on AI and accessibility.

Lastly, thanks to all the members of the web community. If you have ever answered somebody's question online, written a blog post explaining a feature or workaround, or taken the time to file a bug report with a browser – thanks! You've all helped make an industry. Whenever someone asks me, "What's the most impressive thing you've ever done with CSS?", my answer is always the same: "Earned a living."

About the reviewer

Michelle Manemann is a web designer based in the Midwest, USA, with 15 years of experience in EdTech and digital publishing. In everyday life, she leads a team focused on accessibility and inclusive design, while also exploring how AI can reshape the way websites are built and experienced. Beyond creating digital experiences for learners across North America, she hosts a podcast, contributes as a technical reviewer, and designs for her local community. It's her goal in all things to live into the original promise of the web: to broaden opportunity, expand access to information, and bring people together.

Table of Contents

Preface xxv

Free Benefits with Your Book xxx

Part 1: The Fundamentals of Responsive Web Design 1

Chapter 1: The Essentials of Responsive Web Design 3

The browser and device landscape 4

Defining responsive web design 5

Responsive web design in a nutshell • 5

Browser support • 5

Text editors • 7

Tools for software development • 7

Should you be coding with AI? 8

Our first responsive example 9

Our basic HTML file • 9

Taming images • 14

A brief tangent on width/max-width for images • 16

Enter media queries • 17

Breakpoints • 17

Amending the example for a larger screen • 18

The shortcomings of our example • 23

Summary 23

Chapter 2: Writing HTML Markup 25

Getting the start of HTML pages right 27

The doctype • 28

The HTML tag and lang attribute • 28

Character encoding • 28

The forgiving nature of HTML5 markup • 29

A sensible approach to HTML markup • 29

All hail the mighty <a> element 30

New semantic elements in HTML5 31

The <main> element • 32

The <section> element • 33

The <nav> element • 33

The <article> element • 34

The <aside> element • 34

The <header> element • 35

The <footer> element • 35

h1-h6 elements • 35

Headings, sub-headings, and the hgroup element • 36

The <div> element • 37

The <p> element • 37

The <blockquote> element • 37

The <figure> and <figcaption> elements • 38

The <details> and <summary> elements • 39

Create an accordion effect with multiple linked details elements • 42

The <address> element • 42

HTML text-level semantics 43

The <span> element • 43

The <b> element • 43

The <strong> element • 44

The <em> element • 44

The <i> element • 44

Obsolete HTML features 45

How to insert Unicode characters in HTML and CSS • 45

Putting HTML elements to use 46

WCAG accessibility conformance and WAI-ARIA for more accessible web applications ... 48

Web Content Accessibility Guidelines (WCAG) • 48

WAI-ARIA • 49

Taking ARIA further • 49

Embedding media in HTML5 50

Providing alternate media sources 52

Audio and video tags work almost identically 53

Responsive HTML5 video and iframes • 53

How to maintain the aspect ratio • 54

The loading attribute 54

The <dialog> element 55

Summary 58

An exercise 58

Chapter 3: Media and Container Queries 61

The meta viewport tag 64

Why media queries are needed for a responsive web design 66

Basic conditional logic in CSS • 66

Media query syntax 67

Media queries in link tags • 68

Media query on an @import at-rule • 68

Media queries in a CSS file • 69

Inverting media query logic • 69

Combining media queries • 69

A number of different media queries • 70

Everyday media queries • 70

What can media queries test for? • 71

**Features deprecated in CSS Media Queries Level 5 • 72*

Using media queries to alter a design 72
Advanced media query considerations 75
Do media queries affect which files get downloaded or how the page renders? • 75
Should you split media queries into their own files? • 76
Consolidate media queries or scatter them where it suits? • 76
Nesting media and container queries • 78
What about file size? • 78
Media Queries Level 5 79
Interaction media features • 79
The pointer media feature • 80
The hover media feature • 80
User preference media features • 81
The prefers-color-scheme media feature • 81
The prefers-reduced-motion feature • 82
Future possibilities • 83
Range syntax • 83
Container queries 84
Using container queries • 85
Why name the container? • 90
Container relative lengths • 91
Possibilities • 92
Container style queries • 92
Nesting container style queries • 93
Currently only work with CSS custom properties • 94
Summary 95

Chapter 4: Fluid Layout and Flexbox 97

Converting a fixed pixel design into a fluid proportional layout 98
Why do we need flexbox? • 103
Introducing flexbox 103
The bumpy path to flexbox • 104

Getting flexy **104**

Perfect vertically centered text • 104

Offset items • 106

Reverse the order of items • 108

How about if we want them laid out vertically instead? • 109

Column reverse • 110

inline-flex **110**

Flexbox alignment properties **111**

align-items • 113

align-self • 114

Possible alignment values • 115

justify-content • 115

The flex property • 118

Simple sticky footer • 120

Visually re-ordering items **122**

Wrapping with flex **127**

What about gap? • 130

Wrapping up flexbox • 131

Summary **132**

Chapter 5: Layout with CSS Grid 133

What CSS Grid is and the problems it solves **134**

Basic Grid syntax • 134

Grid-specific concepts and terminology • 135

Setting up a Grid Layout **136**

Explicit and implicit • 140

grid-auto-rows and grid-auto-columns • 141

grid-auto-flow • 143

Placing and sizing Grid Layout items **144**

gap • 147

repeat • 147

fr units • 148

Placing items in the grid 148
  span • 149
  dense • 150
  Named grid lines • 151
  grid-template-areas • 154
  auto-fit and auto-fill • 156
  The minmax() function • 158
Shorthand syntax 159
  The grid-template shorthand • 160
  The grid shorthand • 160
  The grid shorthand value – option one • 161
  The grid shorthand value – option two • 161
  The grid shorthand value – option three • 162
  A challenge • 163
Allowing nested elements to take part in the grid 164
The subgrid value 166
Summary 170
Applying what you have learned so far 171

Part 2: Core Skills for Effective Frontend Web Development 173

Chapter 6: CSS Nesting, Layers, Selectors, and More 175

CSS nesting 176
  Understanding the & symbol in nesting • 178
  Writing an ancestor relationship • 179
    Adjacent sibling and data-attribute example • 179
    Lists of nested selectors example • 180
  Nesting conditionals such as media queries • 180
Cascade layers 182
  Naming cascade layers • 184

Where to declare layers • 184

Adding rules to layers • 185

The priority of rules in and outside of layers • 185

Nesting layers inside other layers • 186

The importance of !important • 186

!important in cascade layers • 188

Old browsers • 189

Selectors, units, and capabilities 190

Anatomy of a CSS rule • 190

Pseudo-elements and pseudo-classes • 190

CSS selectors – beyond the normal! 191

CSS attribute selectors • 192

CSS substring matching attribute selectors • 193

The "beginning with" substring matching attribute selector • 193

The "contains an instance of" substring matching attribute selector • 194

The "ends with" substring matching attribute selector • 194

Chaining attribute selectors • 195

CSS structural pseudo-classes 196

The :last-child selector • 197

The nth-child selectors • 197

Understanding what nth rules do • 198

The of selector • 202

nth-based selection in responsive web designs • 203

Combinator selectors – child, next sibling, and subsequent sibling 205

The child combinator • 206

The next sibling • 206

The subsequent sibling • 207

The negation (:not) selector • 207

The empty (:empty) selector • 208

Grouping selectors 209

The :is() functional pseudo-class • 210

The :where() functional pseudo-class • 211

The :has() relational pseudo-class • 212

Limiting the application of selectors with @scope **214**

Targeting elements with @scope to get less specificity • 215

@scope specificity • 216

Using scope inside a style tag • 217

The practicality of @scope • 218

Responsive viewport relative lengths **218**

Using @supports to fork CSS **220**

Feature queries • 220

Using @supports as a proxy for a feature you can't test for • 222

Combining conditionals • 222

Summary **224**

Chapter 7: Web Typography 225

System fonts **226**

The @font-face CSS rule **226**

Using custom fonts with online services • 227

Implementing web fonts with @font-face • 227

Optimizing font loading with font-display **229**

font-display • 230

Consolidating the appearance of two fonts with font-size-adjust and size-adjust • 232

The size-adjust property • 234

Better text alignment with text-box **235**

text-box-trim possible values • 237

text-box-edge possible values • 237

Controlling the flow of text with text-wrap **238**

Variable fonts **240**

font-face changes • 240

How to make use of a variable font • 243

Registered axis • 243

    Custom axis • 244

  Font features • 244

  Fancy numbers with font-variant-numeric • 246

Summary 248

Exercise 248

Chapter 8: CSS Colors 251

RGB color 252

HSL color 253

  Alpha channels • 255

    Alpha with hex • 255

    Alpha with functional notation • 256

Space-separated functional color notation 257

Color concepts and terminology 257

  Color space • 258

  Gamut • 258

The current state of CSS color 260

  Introductory note • 261

  Display-P3 • 261

    The @media (color-gamut) media query • 263

  LCH with the lch() function • 263

  OKLab and OKLch • 266

Fallbacks for older browsers 266

The color-mix() function 267

The contrast-color() function 268

  The original color-contrast() function • 268

  The original color-contrast() function was flawed • 269

    How did this happen? • 269

  The newer, but more limited contrast-color() function • 269

Relative colors with the relative color syntax 270

  Values always have to be numbers • 270

The light-dark() function • 271

Summary 272

Resources 272

Chapter 9: Stunning Aesthetics with CSS 273

Text shadows 274

Multiple text shadows • 275

Box shadows 276

Inset shadow • 276

Multiple box shadows • 277

Understanding spread • 277

Background gradients 278

The linear-gradient notation • 279

Specifying gradient direction • 279

Color stops • 280

Radial background gradients • 281

Handy extent keywords for responsive sizing • 283

Conic gradients • 284

Repeating gradients • 287

Background gradient patterns • 288

Multiple background images 290

Background size • 290

Background position • 291

Background shorthand • 292

High-resolution background images • 293

CSS filters 294

Available CSS filters • 295

Combining CSS filters • 300

A warning on CSS performance 300

CSS clip-path 302

CSS basic shapes • 302

    clip-path with a circle • 303
    clip-path with ellipse • 304
    clip-path with inset • 304
    clip-path with polygon • 305
    clip-path with URL (clipping source) • 306
  The shape() CSS function • 307
    More resources exploring shape() • 309
  Animating clip-path • 310
mask-image 311
mix-blend-mode 314
Summary 314

Chapter 10: Responsive Images 317

Modern image formats 318
AVIF image format 319
WebP image format 319
Image format comparisons 320
Using the picture element to progressively enhance images 321
The inherent problem of responsive images 323
Simple resolution switching with srcset 324
  Advanced switching with srcset and sizes • 325
    Did you say the browser "might" pick one image over another? • 326
  Art direction with the picture element • 326
A practical note on providing alternate image formats and sizes 327
Summary 328

Chapter 11: SVG 329

A brief history of SVG 331
An image that is also a readable web document 332
  The root SVG element • 334
  Namespace • 335

- The title and desc tags • 335
- The defs tag • 335
- The g element • 335
- SVG shapes • 336
- SVG paths • 336

Creating SVGs with popular image-editing packages and services 336

- Saving time with SVG icon services • 337

Inserting SVGs into your web pages 338

- Using an img tag • 338
- Using an object tag • 338
- Inserting an SVG as a background image • 339
  - *A brief aside on data URIs • 339*
  - *Generating image sprites • 340*
- Inserting an SVG inline • 341

Reusing graphical objects from symbols 341

- Inline SVGs allow different colors in different contexts • 344
- Making dual-tone icons that inherit the color of their parent • 344
- Recoloring SVGs with CSS custom properties • 345
- Reusing graphical objects from external sources • 347

Recoloring SVG assets with a mask 348

What you can do with each SVG insertion method (inline, object, background-image, and img) 349

Extra SVG capabilities and oddities 350

- SMIL animation • 350
- Styling an SVG with an external style sheet • 351
- Styling an SVG with internal styles • 352
  - *SVG properties and values within CSS • 353*
- Animating an SVG with CSS • 353
- Animating SVG with JavaScript • 355
  - *A simple example of animating an SVG with GreenSock • 355*
- Using SVGs as filters • 357

A note on media queries inside SVGs • 360
Optimizing SVGs 361
SVG implementation tips 362
Summary 362
Further resources 363

Chapter 12: Transitions, Transformations, and Animations 365

What CSS transitions are and how we can use them 366
The properties of a transition • 368
The transition shorthand property • 369
Transitioning different properties over different periods of time • 370
Understanding timing functions • 370
CSS 2D transforms 373
Scale • 374
Translate • 375
Using translate to center absolutely positioned elements • 376
Rotate • 378
Skew • 379
Matrix • 380
The transform-origin property • 381
CSS 3D transformations 383
The translate3d property • 386
A progressive enhancement example using translate3d • 387
Animating with CSS 390
The animation-fill-mode property • 393
Using @starting-style to transition items entering the DOM 395
But you can't animate them being removed • 397
Transition elements to display: none with transition-behavior • 397
Animating height with the calc-size() function 399
The animation-composition property • 400
Summary 401

Exercises and training 402

Chapter 13: Custom Properties and CSS Functions 403

Working with custom properties 404

Simple theming • 405

Setting and getting custom property values with JavaScript • 407

Custom properties and specificity • 408

Setting a fallback value • 409

env() environment variables • 410

Custom properties and web components • 410

CSS functions 411

calc() • 412

min() • 414

max() • 414

clamp() • 415

Putting it all together 416

Summary 418

Chapter 14: Forms 419

HTML5 forms 420

Understanding the component parts of HTML5 forms • 421

The placeholder attribute • 422

Styling the placeholder text • 422

The required attribute • 423

The autofocus attribute • 424

The autocomplete attribute • 425

The list attribute and the associated datalist element • 426

The pattern input attribute • 428

Styling the input caret with the caret-color CSS property • 429

HTML5 input types 430

The email input type • 430

  The number input type • 432

    Using min and max to create number ranges • 434

    Changing the step increments • 435

  The url input type • 435

  The tel input type • 438

  The search input type • 440

  The color type input • 440

  Date and time • 441

    The date input type • 441

    The month input type • 443

    The week input type • 443

    The time input type • 444

  The range input type • 445

Styling HTML5 forms with CSS 446

  Indicating required fields • 448

  Creating a background fill effect • 452

  New <select> and <option> styling • 453

    Default select styling • 453

  Customizable selects • 455

Summary 456

Part 3: Latest Platform Features and Parting Advice 457

Chapter 15: Cutting-Edge CSS Features 459

The Popover API 459

  Styling the popover • 461

Anchor positioning 464

  Creating an anchor relationship • 465

  Positioning anchored content with position-area • 467

  Positioning with anchor() functions • 469

  Sizing the anchored elements with anchor-size() • 470

Overflow management with position-try • 471
Using position-try • 472
position-try flip tactics • 473
Custom positions with @position-try • 474
Limitations and frustrations of anchor positioning • 475
Scroll-driven animations 478
The animation-timeline property • 480
The animation-range property • 480
Driving the animation with scroll() • 482
View transitions 484
Same-document view transitions • 484
Changing default durations • 486
Every element to transition needs a unique view-transition-name value • 486
Selecting simlilar transitions with view-select-class • 488
Pitfalls with view transitions • 489
Cross-document view transitions • 491
Summary 493

Chapter 16: Bonus Techniques and Parting Advice 495

Truncating text 496
Single-line truncation • 496
Multi-line truncation • 497
Creating horizontal scrolling panels 498
Styling scrollbars • 501
Scrollbar color accessibility • 503
Proprietary -webkit- scrollbar styling • 504
CSS Scroll Snap • 505
The scroll-snap-type property • 505
The scroll-snap-align property • 506
The scroll-snap-stop property • 506

Smooth scrolling with CSS scroll-behavior • 507

Rounded gradient borders with semi-transparent backgrounds • 508

Parting advice **511**

Getting designs in the browser as soon as possible • 511

Testing on real devices • 512

Embracing progressive enhancement • 512

Defining a browser support matrix • 513

Functional parity, not visual parity • 513

Choosing the browsers to support • 514

Tiering the user experience • 514

Avoiding CSS frameworks in production • 515

Hiding, showing, and loading content across viewports • 515

Validators and linting tools • 516

Performance • 517

Performance tools • 518

The next big things • 520

Summary 521

Other Books You May Enjoy **525**

Index **529**

Preface

When I wrote the first edition of this book in 2011 and 2012, it was by no means certain that responsive web design would become the *de facto* approach for web design. Here we are, over a decade and four further editions of the book later, and the popularity and utility of the approach show no sign of abating.

There's a saying that is apt for the authors of technical books: "When one person teaches, two people learn." That's certainly been the case each time I have written a new edition. I've learned so much more covering these topics than I imagined I would. In my day-to-day work, I already find myself coming back to certain chapters and sections when trying to refresh my memory on how to do something new I have subsequently forgotten! I hope these pages prove as resourceful for you!

Thinking back on the contents of the first edition, it also strikes me just how capable the technologies we have at our disposal have become. If you are an old hand at the web development game, feel free to jump right into some of the newest topics, such as view transitions or scroll-driven animations, or take a look at expanded sections on web typography or CSS nesting – I'd be shocked if you didn't come away at least a bit excited. Many of the things now trivial would have blown my mind as I sat down to write the first edition well over a decade ago.

I won't waste any more of your precious time except to say thank you for taking the time to read this book. I hope you enjoy this latest edition and take plenty from it. Please reach out to me with your thoughts, both good and bad, and any questions that arise. They will naturally inform the content of any future editions.

Finally, while the publishers will mention it again in a moment, if you do enjoy it, please consider adding a review on Amazon or your book store of choice. From a commercial point of view, it really does help with sales. From a personal perspective, it's really lovely to read them and know that people around the world are making use of something I've spent so long working on.

It should go without saying that if you don't enjoy it, please keep your opinions to yourself!

Who this book is for

Are you a full stack or backend developer who needs to improve their frontend skills? Perhaps you work on the frontend and you need a definitive overview of all modern HTML and CSS has to offer? Maybe you have done a little website building but you need a deep understanding of responsive web designs and how to achieve them? This is the book for you!

All you need to take advantage of this book is a working understanding of HTML and CSS. No JavaScript knowledge is needed.

What this book covers

Chapter 1, *The Essentials of Responsive Web Design*, is a whistle-stop tour of the key ingredients in coding a responsive web design.

Chapter 2, *Writing HTML Markup*, covers all the semantic elements of HTML5, text-level semantics, and considerations of accessibility. We also cover how to insert media such as video into our pages and how to use the new `dialog` element.

Chapter 3, *Media and Container Queries*, covers everything you need to know about CSS media and container queries: their capabilities, the syntax, and the various ways you can wield them.

Chapter 4, *Fluid Layout and Flexbox*, covers how to convert designs into proportional layouts and provides a thorough exploration of Flexbox layouts.

Chapter 5, *Layout with CSS Grid*, is a deep dive into the two-dimensional layout system of CSS grid, and how to make use of the newer subgrid property.

Chapter 6, *CSS Nesting, Layers, Selectors, and More*, gets us to grips with the endless possibilities of CSS selectors and feature queries, as well as how to organize your style sheets with CSS layers and write them using CSS nesting.

Chapter 7, *Web Typography*, is your one-stop resource for getting up to speed on all things web typography – not just how to display fonts but fine details such as loading optimizations and new wrapping capabilities.

Chapter 8, *CSS Color*, covers the latest developments in expressing colors and color manipulation, including Lab/LCH and P3 color, as well as how to wield functions such as `color-contrast` and `color-mix`.

Chapter 9, *Stunning Aesthetics with CSS*, covers CSS filters, box shadows, linear and radial gradients, multiple backgrounds, and how to target background images to high-resolution devices.

Chapter 10, *Responsive Images*, covers how to make use of new image formats such as AVIF and WebP, and how to make the most relevant version of an image available to the browser.

Chapter 11, *SVG*, teaches us everything we need to use SVG graphics inside documents and as background images, as well as how to interact with them using JavaScript.

Chapter 12, *Transitions, Transformations, and Animations*, gets our CSS moving as we explore how to make interactions and animations using CSS.

Chapter 13, *Custom Properties and CSS Functions*, covers the many benefits of custom properties, then moves on to the utility of new CSS functions such as `min()`, `max()`, and `clamp()`.

Chapter 14, *Forms*, shows us how the latest HTML5 and CSS features make interacting with forms easier than ever before. Many HTML input types give you easy access to the relevant software keyboard on mobile devices.

Chapter 15, *Cutting-Edge CSS Features*, covers how to use view transitions to move the UI between different states, how to apply scroll-driven animations with no JavaScript, and understanding CSS anchor positioning for popovers, menus, and more.

Chapter 16, *Bonus Techniques and Parting Advice*, explores the essential considerations before embarking on a responsive web design and also provides a few last-minute nuggets of wisdom to aid you in your responsive quest.

Free downloadable bonus content

To further support your learning, we've made two bonus pieces available to download for free. These have been specially prepared by our technical reviewer, *Michelle Manemann*, to provide deeper coverage of topics that extend beyond the main text.

- **Accessibility in Responsive Web Design**: A practical resource you'll want to reference throughout the book whenever accessibility considerations come up, with techniques and checklists you can apply directly to your projects.
- **Generative AI in Responsive Web Design**: A forward-looking guide that helps you evaluate when and how AI-generated code can support your frontend workflow. Use it as a companion when experimenting with the AI-related discussions in the later chapters.

You can access both appendices online here (`https://landing.packtpub.com/responsive-web-design/`) and keep them as a handy reference alongside the core chapters.

To get the most out of this book

- You’ll need a text editor such as Sublime Text, Neovim, or VS Code
- You’ll need a modern browser such as Firefox, Edge, Safari, or Chrome
- You’ll need an appreciation of mediocre jokes and popular 90s film references

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/Responsive-Web-Design-with-HTML5-and-CSS-Fifth-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/gbp/9781837028238`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter/X handles. For example: “We can fix that prior problem easily by adding this snippet in `<head>`.”

A block of code is set as follows:

```
img {
    max-width: 100%;
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
img {
    max-width: 100%;
    display: inline-flex;
}
```

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes, also appear in the text like this. For example: “When writing HTML, you will typically be **marking up** or writing content inside a series of **tags** or **elements**.”

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packt.com/`.

Free Benefits with Your Book

This book comes with free benefits to support your learning. Activate them now for instant access (see the "*How to Unlock*" section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:

PDF and ePub Copies	Next-Gen Web-Based Reader
EPUB PDF Free PDF and ePub versions	Next-Gen Reader
Access a DRM-free PDF copy of this book to read anywhere, on any device. Use a DRM-free ePub version with your favorite e-reader.	**Multi-device progress sync**: Pick up where you left off, on any device. **Highlighting and notetaking**: Capture ideas and turn reading into lasting knowledge. **Bookmarking**: Save and revisit key sections whenever you need them. **Dark mode**: Reduce eye strain by switching to dark or sepia themes.

Unlock this book's exclusive benefits now

UNLOCK NOW

Scan this QR code or go to `https://packtpub.com/unlock`, then search this book by name.

***Note:** Keep your purchase invoice ready before you start.*

Share your thoughts

Once you've read *Responsive Web Design with HTML5 and CSS — Fifth Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Part 1

The Fundamentals of Responsive Web Design

This first section is described as fundamental, as it is the section on which the other two build. In *Chapter 1*, *The Essentials of Responsive Web Design*, we will create our first fully responsive website. In *Chapter 2*, *Writing HTML Markup*, we are going to understand how to express meaning through markup. Avoid understanding HTML elements at your peril! Many problems in web development, even among seasoned developers, can be avoided simply by using structured and semantic markup. Finally, in *Chapter 3*, *Media and Container Queries*, we will learn how to encapsulate styles that only apply in certain eventualities, functionality that's a cornerstone of responsive web design. The final two chapters, *Fluid Layout and Flexbox*, and *Layout with CSS Grid*, conclude this first part, covering the two most powerful layout methods available to us in modern web design, Flexbox and Grid.

If you feel these are topics you already have a strong grasp of, go right ahead and skip to the next part. Otherwise, take the time to deeply understand these fundamentals before moving on. With the fundamentals understood, everything that comes later – in this book, and in your web adventures – will be so much easier to achieve.

1

The Essentials of Responsive Web Design

When the first edition of this book came out in 2012, responsive web design was a new and exciting possibility to address the needs of the ever-growing list of devices that could access the internet. Over a decade later, as I write this in 2025, it's simply the de facto standard. If you're building a website or web application and it isn't responsive, you're probably doing it wrong!

This first chapter serves as a quick and basic refresher on building out an extremely simple web design, responsively. By the end, we will have covered everything needed to author a fully responsive web page.

You might be wondering, why do we need the other chapters then? By the end of this chapter, that should be apparent too.

Here's what we will cover in this first chapter:

- The ever-evolving browser and device landscape
- Defining responsive web design
- Setting browser support levels
- A brief discussion on development tools and text editors
- Our first responsive example: a simple HTML5 page
- The viewport meta tag
- Fluid images
- Writing CSS3 media queries to make pages adapt

- The shortfalls in our basic example
- Why our journey has only just begun

Are you sitting comfortably? Then we will begin!

The browser and device landscape

At the turn of the century, it was reasonable to build websites with fixed widths. The device landscape was a whole lot more limited, so the expectation was that all end users would get a fairly consistent experience. This fixed width (typically 960 px wide or thereabouts) wasn't too wide for laptop screens, and users with large-resolution monitors merely had an abundance of space on either side.

But in 2007, Apple's iPhone ushered in the first truly usable browsing experience on a phone, and the way people accessed and interacted with the web changed forever.

In the first edition of this book, published in early 2012, the following was noted about the percentage of total browser usage, globally, by device type:

> *...in the 12 months from July 2010 to July 2011, global mobile browser use had risen from 2.86 to 7.02 percent...*

By September 2019, writing the third edition, using StatCounter, mobile was a whopping 51.11%, desktop was 45.18%, and tablet was 3.71%.

Now, in February 2025, `gs.statcounter.com` tells us mobile is now 62.71% and desktop 35.4%. Tablet is now only 1.88%.

The indisputable fact is that the number of people using smaller-screen devices to view the internet is growing at an ever-increasing rate, while at the other end of the scale, 40-inch ultrawide displays are now also commonplace (along with various tablet and console devices). There is now more of a difference between the smallest screens browsing the web and the largest than ever before.

Thankfully, there is a solution to this ever-expanding browser and device landscape. A responsive web design, built with HTML and CSS, allows a website to "just work" across multiple devices and screens. It enables the layout and capabilities of a website to respond to its environment (screen size, input type, device/browser capabilities).

Defining responsive web design

The term **responsive web design** was coined by Ethan Marcotte in 2010. In his seminal article, *Responsive Web Design* (`http://www.alistapart.com/articles/responsive-web-design`), he consolidated three existing techniques (flexible grid layout, flexible images/media, and media queries) into one unified approach and named it responsive web design.

Responsive web design in a nutshell

To attempt to put the philosophy of responsive web design in a "nutshell," I would say it's the presentation of web content in the most relevant format for the viewport and device accessing it.

In its infancy, it was typical for responsive design to be implemented by starting with a fixed-width desktop design before trying to scale the design down as needed for smaller screens. However, processes evolved and it became apparent there was a better way. Namely, everything from design to content management and development worked better when starting with the smallest screens first, and then "progressively enhancing" the design and content for larger screens and/or more capable devices. If the term "progressive enhancement" makes no sense right now, fear not. We'll be talking about that again in a moment.

Before we get into things fully, there are a few subjects I'd like to address and get squared away before we continue: browser support, text editors, and tooling.

Browser support

The sheer volume of disparate devices that access the web means most people understand the need for technical solutions that cater to most devices.

The popularity and ubiquity of responsive web design usually make the approach an easy sell to clients and stakeholders. Nowadays, most people have some idea of what responsive web design is about, even if that understanding amounts to little more than "a website that looks good and works on phones as well as computers."

However, one question that almost always comes up when starting a responsive design project is that of browser support. With so many browser and device variants, it's not always pragmatic to support every single browser permutation fully. Perhaps time is a limiting factor, perhaps money. Perhaps both.

Typically, the older the browser, the greater the work and code required to achieve feature or aesthetic parity with modern, "evergreen" browsers such as Firefox, Chrome, or Edge. Evergreen, in this context, means browsers that automatically upgrade themselves every couple of months to include the latest web platform features. But as some browsers, such as Safari in iOS, only get updated a few times a year, you will need to be as pragmatic as possible about the features you employ.

By the same token, we are going to practice progressive enhancement. In essence, starting with a functional and accessible website for the most basic browsers, which will get progressively enhanced with features for more capable browsers. It should be a very rare occasion indeed that you are forced to create a website that isn't at least functional on an old browser or device.

Ultimately, the only browser statistics that really matter are yours.

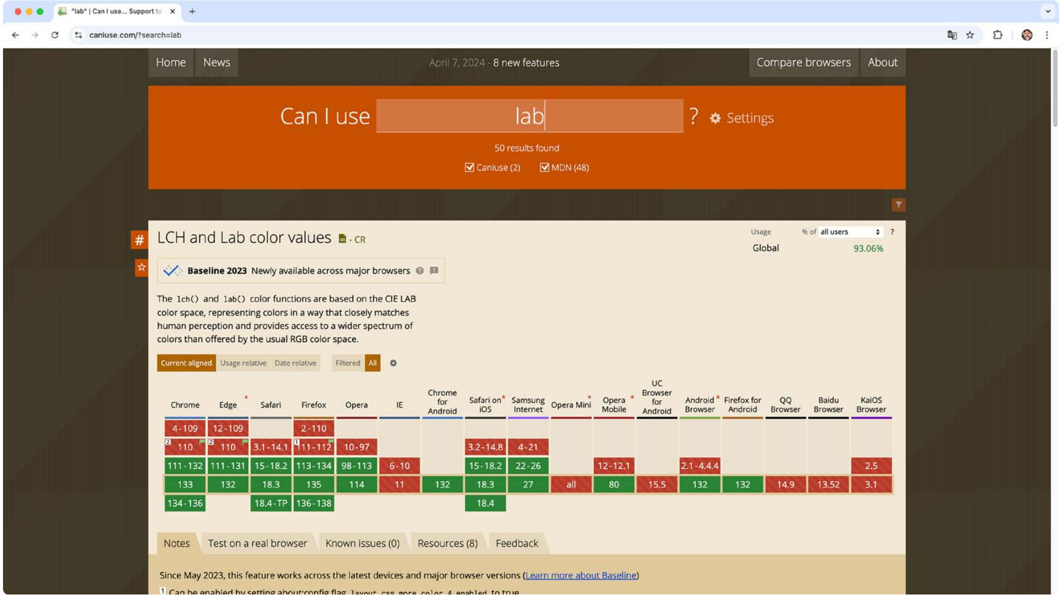
> If you are working on a greenfield project, where there is no existing browser usage data, you can at least think about the demographics of your target audience and make some broad assumptions about likely devices/browsers based on those demographics.

Before considering any web project, it makes sense to decide, in advance, what platforms you need to fully support and which you are happy to concede visual/functional anomalies to.

For example, if you're unlucky enough to have 25% of your website visitors stuck using old versions of iOS (for example), you'll need to consider what features that browser supports and tailor your solution accordingly.

To this end, if you aren't already, become familiar with websites such as `http://caniuse.com`. *Can I Use* provides a simple interface for establishing browser support for each web platform feature.

Generally speaking, when starting a project, as a simple and broad way to determine what browsers to support, I apply the following crude piece of logic: if the cost of developing and supporting browser `X` is more than the revenue/benefit created by the users on browser `X`, don't develop specific solutions for browser `X`.


"lab" | Can I use... Support ta
caniuse.com/?search=lab
Home
News
April 7, 2024 - 8 new features
Compare browsers
About
Can I use
lab
?
Settings
#
☆
Usage
% of
all users
?
Current aligned
Usage relative
Date relative
Filtered
All
Notes
Test on a real browser
Known issues (0)
Resources (8)
Feedback

50 results found

☑ Caniuse (2) ☑ MDN (48)

LCH and Lab color values - CR

Global 93.06%

Baseline 2023 Newly available across major browsers

The `lch()` and `lab()` color functions are based on the CIE LAB color space, representing colors in a way that closely matches human perception and provides access to a wider spectrum of colors than offered by the usual RGB color space.

Chrome	Edge *	Safari	Firefox	Opera	IE	Chrome for Android	Safari on iOS *	Samsung Internet	Opera Mini *	Opera Mobile *	UC Browser for Android	Android * Browser	Firefox for Android	QQ Browser	Baidu Browser	KaiOS Browser
4 - 109	12 - 109		2 - 110													
[2] 110	[2] 110	3.1 - 14.1	[1] 111 - 112	10 - 97			3.2 - 14.8	4 - 21								
111 - 132	111 - 131	15 - 18.2	113 - 134	98 - 113	6 - 10		15 - 18.2	22 - 26		12 - 12.1		2.1 - 4.4.4				2.5
133	132	18.3	135	114	11	132	18.3	27	all	80	15.5	132	132	14.9	13.52	3.1
134 - 136		18.4 - TP	136 - 138				18.4									

Since May 2023, this feature works across the latest devices and major browser versions (Learn more about Baseline)

[1] Can be enabled by setting about:config flag `layout.css.more_color_4.enabled` to true

Figure 1.1: Can I Use provides browser support data for every web platform feature

Text editors

It makes no difference what tool you use to write your code. All you actually need is something that enables you to type HTML, CSS, and JavaScript. Whether your preference is Sublime Text, Vim, Emacs, Nova, Visual Studio Code, or Notepad – it matters little. Just use what works best for you.

Tools for software development

Similarly, there are no requisite tools that are essential to get responsive web design out of the door. That said, you should be aware that there are many freely available tools to negate many of the manual and time-intensive tasks of building websites. CSS post-processors such as PostCSS can automate horrible and thankless jobs such as CSS vendor prefixing, an old requirement that necessitated writing multiple versions of a property for each browser engine, and poly-filling new CSS features, where newer syntaxes can be made to work on older browsers. Linting and validation tools can check your HTML, JavaScript, and CSS code as you work, eliminating many time-wasting typos or syntax-induced errors.

More recently, code formatters have become popular. Tools such as Prettier, for example, automatically format your code with indentation and spacing when you save. None of these tools are essential but they may afford you some benefits.

New tools come out constantly and they are continually evolving. Therefore, while some relevant and beneficial tools will be mentioned by name as we go, be aware that something better may be just around the corner. Hence, we won't be relying on anything other than standards-based HTML and CSS in our examples. You should, however, use whatever tools you can bring to bear to produce your frontend code as quickly and reliably as possible.

With that out of the way, without further ado, let's get our first responsive web design made!

Should you be coding with AI?

I am often asked if it still makes sense to be learning to code "by hand" when there are so many AI-powered tools to write code for us. By extension, is it even worth learning about concepts and approaches such as responsive web design? Surely, we can just feed a service such as ChatGPT, Claude, DeepSeek, or any other of the burgeoning AI-powered tools our requirements with text, speech, or images and copy and paste in the result?

Well, the answer is both yes and no.

While I occasionally find these tools useful to reinforce my understanding, knowing what I want, and how things conceptually work, is fundamental to getting good results from such tools. It enables me to validate the responses and gives me a solid "sniff test" for whether something seems useful or not.

Imagine you have been a house builder for some years, and you now know how to build a house and build it well. After so many years mastering your trade, you are established enough to employ more junior builders to work for you. Will these builders automatically meet your standards of craftsmanship? Wouldn't it be prudent to check the quality of their workmanship to ensure it meets your own high standards if the homes you are building, and therefore your own reputation, are to stand the test of time?

This is generally how I feel about such tools.

Understanding how a house is built lets you understand the true value of tools that can help you build it faster, stronger, and more beautifully.

The other practical consideration of AI is that it is often based on outdated training data. As such, it is unlikely to have a solid grasp of the latest platform features as, at least at the time I write this, AI and large language models are only ever mining the past. They can't, despite what the AI hype might try and convince you, actually learn new things for themselves.

Our first responsive example

In the introduction, I promised that by the end of this chapter, you would know all you needed to build a fully responsive web page. So far, I've just been talking around the issue at hand. It's time to walk the walk.

Code samples

You can download all the code samples from this book by visiting `rwd.education`. It's worth knowing that where individual examples are built up throughout a chapter, there will typically be a "start" and "end" version for each. The start contains just the essentials to start following along. The end contains the completed exercise/example.

Our basic HTML file

We will start with a simple HTML5 structure. Don't worry at this point about what each of the lines does, especially the content of `<head>`, as we will cover that in detail in *Chapter 2*, *Writing HTML Markup*.

For now, concentrate on the elements inside the `<body>` tag. There, we have a few divs, a graphic for a logo, some text, and a list of items. Although you can see more of that content in the screengrabs, a shorter version of the code follows. For brevity, I have removed the paragraphs of text, as we only need to concern ourselves with the core structure.

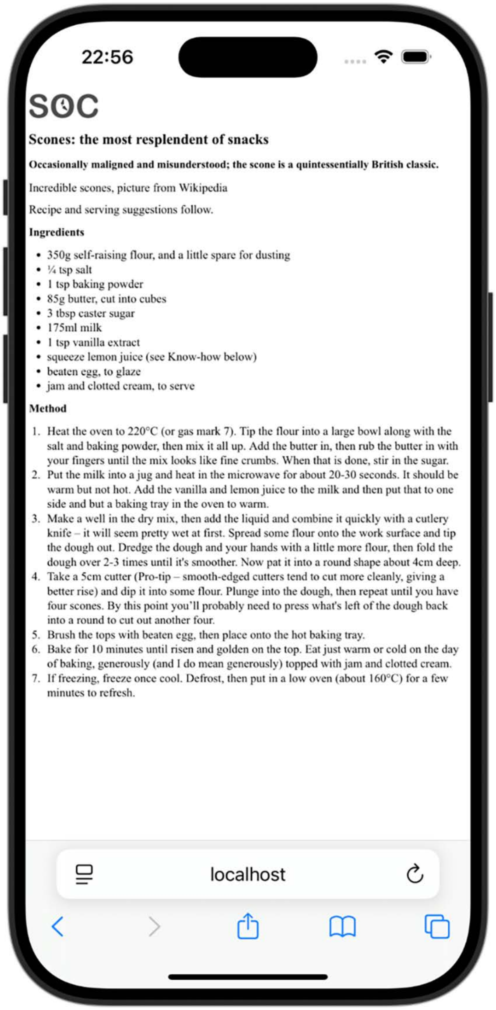
However, what you should know is that the text is a recipe and description of how to make scones: a quintessentially British dessert.

Remember, if you want to get your hands on the full HTML file, you can download the example code from the `rwd.education` website.

```
<!DOCTYPE html>
<html class="no-js" lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Our first responsive web page with HTML5 and CSS3</title>
        <meta
            name="description"
            content="A basic responsive web page – an example from
            Chapter 1"
        />
        <link rel="stylesheet" href="css/styles.css" />
    </head>
    <body>
        <div class="Header">
            <a href="/" class="LogoWrapper">
                <img src="img/SOC-Logo.png" alt="Scone O'Clock logo"
            /></a>
            <h1 class="Strap">Scones: the most resplendent of snacks</h1>
        </div>
        <div class="IntroWrapper">
            <h2 class="IntroText">
                Occasionally maligned and misunderstood; the scone is a
                quintessentially British classic.
            </h2>
            <div class="MoneyShot">
                <p class="ImageCaption">
                    Incredible scones, picture from Wikipedia
                </p>
            </div>
        </div>
        <p>Recipe and serving suggestions follow.</p>
        <div class="Ingredients">
            <h3 class="SubHeader">Ingredients</h3>
            <ul><!--list of ingredients here --></ul>
        </div>
        <div class="HowToMake">
            <h3 class="SubHeader">Method</h3>
            <ol class="MethodWrapper"><!—steps ◊ </ol>
        </div>
    </body>
</html>
```

By default, web pages are inherently flexible. If I open the example page, even as it is at this point, with no special work done to make it responsive, and resize the browser window, the text re-flows as needed.

What about on different devices? Again, with no CSS whatsoever added to the page, this is how that renders on an iPhone 16 Pro:



SOC

Scones: the most resplendent of snacks

Occasionally maligned and misunderstood; the scone is a quintessentially British classic.

Incredible scones, picture from Wikipedia

Recipe and serving suggestions follow.

Ingredients

- 350g self-raising flour, and a little spare for dusting
- ¼ tsp salt
- 1 tsp baking powder
- 85g butter, cut into cubes
- 3 tbsp caster sugar
- 175ml milk
- 1 tsp vanilla extract
- squeeze lemon juice (see Know-how below)
- beaten egg, to glaze
- jam and clotted cream, to serve

Method

1. Heat the oven to 220°C (or gas mark 7). Tip the flour into a large bowl along with the salt and baking powder, then mix it all up. Add the butter in, then rub the butter in with your fingers until the mix looks like fine crumbs. When that is done, stir in the sugar.
2. Put the milk into a jug and heat in the microwave for about 20-30 seconds. It should be warm but not hot. Add the vanilla and lemon juice to the milk and then put that to one side and but a baking tray in the oven to warm.
3. Make a well in the dry mix, then add the liquid and combine it quickly with a cutlery knife – it will seem pretty wet at first. Spread some flour onto the work surface and tip the dough out. Dredge the dough and your hands with a little more flour, then fold the dough over 2-3 times until it's smoother. Now pat it into a round shape about 4cm deep.
4. Take a 5cm cutter (Pro-tip – smooth-edged cutters tend to cut more cleanly, giving a better rise) and dip it into some flour. Plunge into the dough, then repeat until you have four scones. By this point you'll probably need to press what's left of the dough back into a round to cut out another four.
5. Brush the tops with beaten egg, then place onto the hot baking tray.
6. Bake for 10 minutes until risen and golden on the top. Eat just warm or cold on the day of baking, generously (and I do mean generously) topped with jam and clotted cream.
7. If freezing, freeze once cool. Defrost, then put in a low oven (about 160°C) for a few minutes to refresh.

Figure 1.2: Not pretty, but by default all web pages are inherently flexible

As you can see, it's rendering, but like a desktop page shrunken down to fit the space available. The reason for that is that iOS renders web pages at 980 px wide by default and shrinks them down into the "viewport."

Before responsive design was “a thing,” it was commonplace to see websites render like that on an iPhone. Nowadays, thanks to the ubiquity of responsive web design, they are as rare as rocking horse droppings!\

> The area of a browser window that a web page is allowed to be viewed in is known technically as the viewport. To be clear, the viewport area excludes the browser toolbars, URL bar, and so on. From now on, we will generally use this more accurate term.

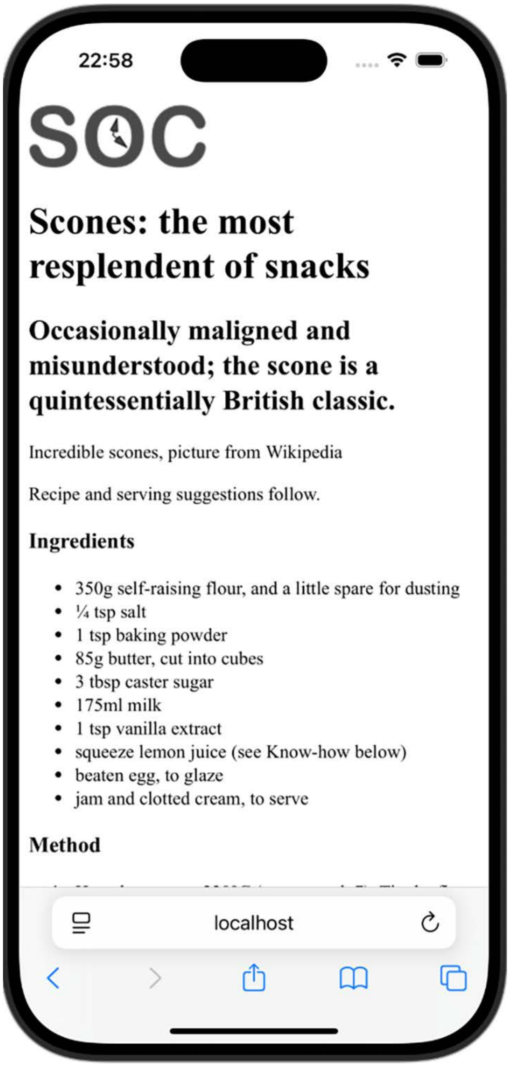
We can make the page more mobile-friendly by adding this snippet in `<head>`:

```html
<meta name="viewport" content="width=device-width,initial-scale=1.0" />
```

This viewport `meta` tag is the non-standard but de facto way of telling the browser how to render the page. Although introduced to the web by Apple, rather than a standard process, it remains essential for responsive web design. We will cover the `meta` tag and its various settings and permutations in *Chapter 3, Media and Container Queries*.

For now, you just need to know that, in this case, our viewport `meta` tag is effectively saying “make the content render at the width of the device.”

In fact, it’s probably easier to just show you the effect this line has on applicable devices:



SOC

Scones: the most resplendent of snacks

Occasionally maligned and misunderstood; the scone is a quintessentially British classic.

Incredible scones, picture from Wikipedia

Recipe and serving suggestions follow.

Ingredients

- 350g self-raising flour, and a little spare for dusting
- ¼ tsp salt
- 1 tsp baking powder
- 85g butter, cut into cubes
- 3 tbsp caster sugar
- 175ml milk
- 1 tsp vanilla extract
- squeeze lemon juice (see Know-how below)
- beaten egg, to glaze
- jam and clotted cream, to serve

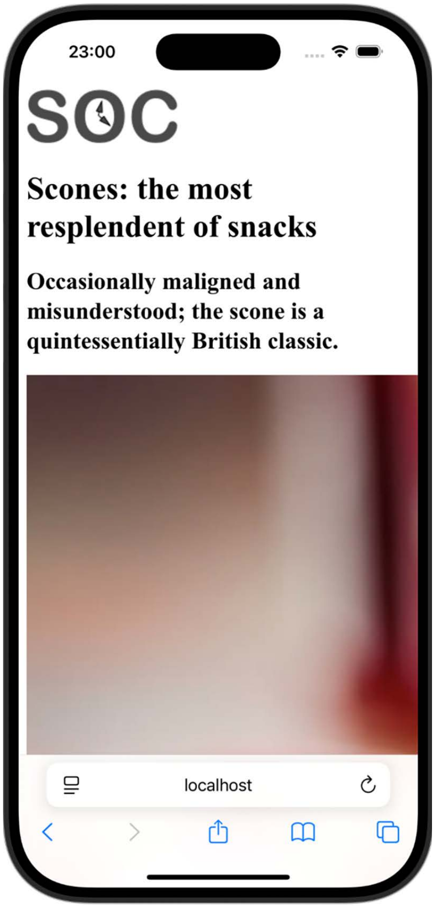
Method

Figure 1.3: With just one line added, already things are improving dramatically

Great! Another snag fixed; the text is now rendering and flowing at a more “native” size. Let’s move on to images.

Taming images

They say a picture speaks a thousand words. All this writing about scones in our sample page and there's no image of the beauties! I'm going to add in an image of a scone near the top of the page – a sort of "hero" image to entice users to read the page.



Scones: the most resplendent of snacks

Occasionally maligned and misunderstood; the scone is a quintessentially British classic.

Figure 1.4: There is a line or two of CSS that's always needed to make images appear a sensible size

Oh! That nice big image (2000 px wide) is forcing our page to render more than a little wonky. We clearly need to fix that.

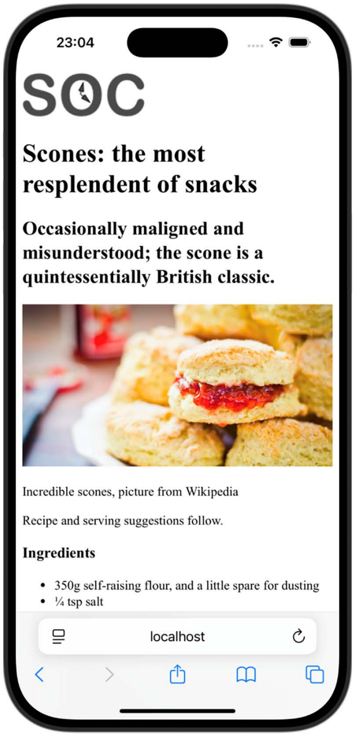
Ideas? Well, we could add a fixed width to the image via CSS, but the problem there is that we want the image to scale to different screen sizes. For example, in CSS, our iPhone 16 Pro is 402 px wide by 874 px high. If we set a width of 402 px to that image, what happens if a user rotates the screen? On this device, the 402 px-wide viewport is now 874 px wide. Thankfully, it's pretty easy to achieve fluid images with a single line of CSS.

I'm going to create the `css/styles.css` CSS file now that's already linked in the head of the HTML page.

In our blank `styles.css` file, here is the first thing I'm adding. Ordinarily, I'd be setting a few other defaults, and we'll discuss those defaults in later chapters, but for our purposes I'm happy to open with just this:

```css
img {
    max-width: 100%;
}
```

With that file saved and the page refreshed, we see something more akin to what we might expect.



Scones: the most resplendent of snacks

Occasionally maligned and misunderstood; the scone is a quintessentially British classic.

Incredible scones, picture from Wikipedia

Recipe and serving suggestions follow.

Ingredients

- 350g self-raising flour, and a little spare for dusting
- ¼ tsp salt

Figure 1.5: With a little CSS, our images will never exceed their bounds

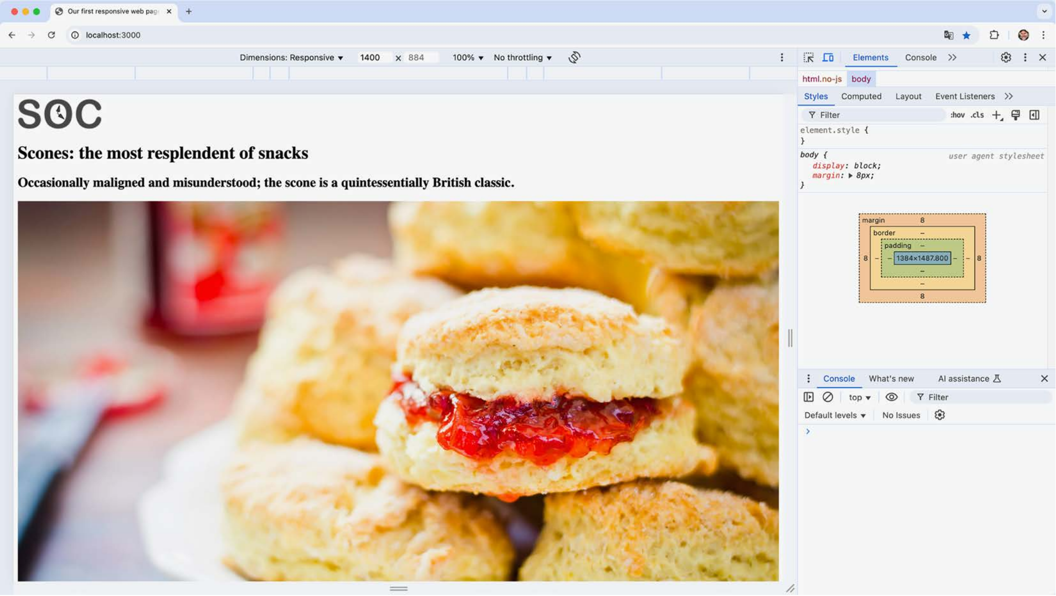
All this max width-based rule does is stipulate that all images should grow to a maximum of 100% of their size. Where a containing element (such as body or a `div` it sits within) is less than the full intrinsic width of the image, the image will simply scale up to display as large as it can within that constraint.

A brief tangent on width/max-width for images

To make images fluid, you could also use the more widely used `width` property – for example, `width: 100%` – but this has a different effect. When a property of `width` is used, then the image will be displayed at that width, relative to its container if using percentages, regardless of its own inherent size. The result in our example would be that the logo (also an image) would stretch beyond its intrinsic size to fill 100% of its container. With a container far wider than the image, as is the case with our logo, this leads to a massively oversized image.

Excellent. Everything is now laid out as expected. No matter the viewport size, nothing is overflowing the page horizontally.

However, if we look at the page in larger viewports, the basic styles start to get both literally and figuratively stretched. Take a look at the example page at a size around 1400 px:



Our first responsive web page
localhost:3000
Dimensions: Responsive ▾ 1400 × 884 100% ▾ No throttling ▾
Elements
Console
html.no-js
body
Styles
Computed
Layout
Event Listeners
Filter
:hov .cls
margin 8
border –
padding –
1384×1487.800
8
What's new
AI assistance
top ▾
Default levels ▾
No Issues

SOC

Scones: the most resplendent of snacks

Occasionally maligned and misunderstood; the scone is a quintessentially British classic.

```
element.style {
}
body {                    user agent stylesheet
   display: block;
   margin: ▶ 8px;
}
```

Figure 1.6: We clearly need to fix the size of this image for larger viewports

Oh dear! In fact, at even around 800 px wide it's starting to suffer. Around this point, it would be handy if we could rearrange a few things. Maybe resize the image and position it off to one side. Perhaps alter some font sizes and background colors of elements.

Thankfully, we can achieve all this functionality quite easily by employing CSS media queries to bend things to our will.

Enter media queries

As we have established, somewhere beyond the 800-px-wide point, our current layout starts to look stretched. We'll use CSS media queries at this point to adjust the layout depending on the screen width. We will cover media queries in greater depth in *Chapter 3*, which is the inventively titled *Media and Container Queries*, but for now, all you need to appreciate is that media queries are directives in CSS that allow us to isolate CSS rules to certain environment conditions – the size of the screen in this instance.

Breakpoints

Before we proceed, it's worth familiarizing you with the term "breakpoint."

The term "breakpoint" is web developer vernacular to define a viewport width or height at which a responsive design should change significantly.

When people first started making use of media queries, it was common to see designs built with specific breakpoints to cater to the popular devices of the day. At the time, it was typically iPhone (320 px x 480 px) and iPad (768 px x 1024 px) devices.

That practice was a bad decision then, and it would be an even worse one now. The problem is that doing that means a design is catered to specific screen sizes. We want a responsive design – something that is agnostic of the screen size viewing it, responding to any size viewport it finds itself in, not something that only looks its best at specific sizes.

Use a breakpoint if your design, visually, needs to change at a certain point, but not to cater to a specific device!

For the purpose of whipping our basic example into shape, however, we will concentrate on just one type of media query: a minimum-width media query. CSS rules within this type of media query only get applied if the viewport is or exceeds a certain width. The exact minimum width can be specified using a raft of different-length units including percent, `em`, `rem`, `vw`, and `px`. In CSS, a minimum-width media query is written like this:

```
@media screen and (min-width: 800px) {
    /* styles /*
}
```

The `@media` directive tells the browser we are starting a media query, the `screen` part (declaring "screen" is technically not needed in this situation, but we will deal with that in detail in *Chapter 3*) tells the browser these rules should be applied to all screen types, and we then have the `and` keyword, which chains together another set of conditionals, which in this case is the `(min-width: 800px)`. That tells the browser that the rules should also be limited to all viewports at least 800 px wide.

I believe it was Bryan Rieger, `http://www.slideshare.net/bryanrieger/rethinking-the-mobile-web-by-yiibu`, who first wrote that:

> *The absence of support for @media queries is in fact the first @media query.*

What he meant by that is that the first rules we write, outside of a media query, should be our starter, or "base," rules for the most basic devices, which we then enhance for more capable devices and larger screens.

That is what we are doing in this example. The basic styles are written first. It is only when we need to do something different that we introduce a media query.

This approach also facilitates a "smallest screen first" mentality and allows us to progressively layer on detail as and when the design needs to change for bigger screens.

Amending the example for a larger screen

We've already established that our design is starting to suffer at around 800 px width.

Therefore, let's mix things up a little by way of a simple example of how we can lay things out differently at different viewport sizes.

First off, we will stop that main “hero” image from getting too big and keep it over on the right. Then the intro text can sit to the left.

We will then have the main portion of text (the “method” that describes how to make the scones) on the left below, with a small boxed-out section detailing the ingredients over on the right.

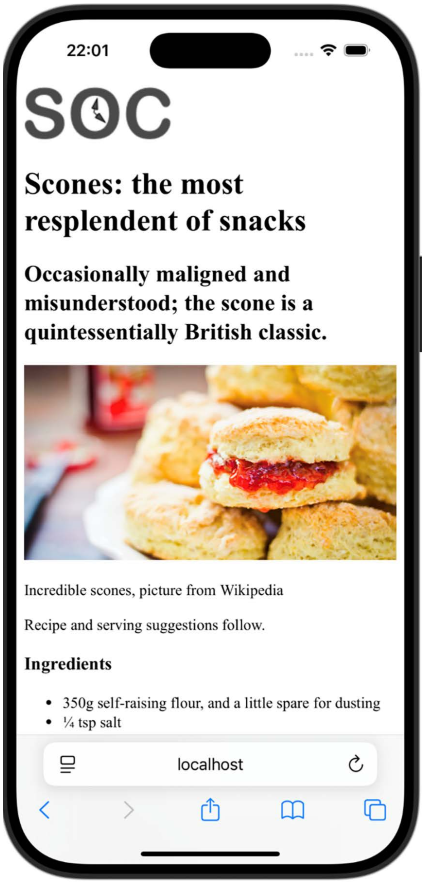
All these changes can be achieved relatively simply by encapsulating these specific styles within a media query.

There are some further visual embellishments that don’t add to the understanding of what’s happening responsively, hence I have omitted them here, but if you’d like to view the relevant code, download the chapter code at `http://rwd.education`.

Here are the layout styles that were added:

```
@media screen and (min-width: 800px) {
    body {
        border-left: 4px solid #f9f9f9;
        border-right: 4px solid #f9f9f9;
        padding: 1rem 2rem;
    }

    .IntroWrapper {
        display: flex;
        gap: 0 20px;
        flex: 1 1 auto;
        align-items: center;
    }

    .MoneyShot,
    .IntroText {
        margin: 0;
        flex: 1 1 50%;
    }

    .MoneyShotImg {
        filter: drop-shadow(0 0 20px #0008);
        border: 0;
    }

    .IntroText {
        padding: 0.5rem;
        font-size: 2.5rem;
        text-align: left;
        position: relative;
    }

    .Ingredients {
        font-size: 0.9rem;
        float: right;
        padding: 1rem;
        margin: 0 0 0.5rem 1rem;
        border-radius: 3px;
        background-color: #ffffdf;
        border: 9px solid #debb71;
    }

    .Ingredients h3 {
        margin: 0;
    }
}
```

That wasn't too bad, was it? With only minimal code, we have built a page that responds to the viewport size and offers a preferable layout as needed. By adding just a few more styles, things look even easier on the eye. With those in place, our basic responsive page now looks like this on an iPhone:



Scones: the most resplendent of snacks

Occasionally maligned and misunderstood; the scone is a quintessentially British classic.

Incredible scones, picture from Wikipedia

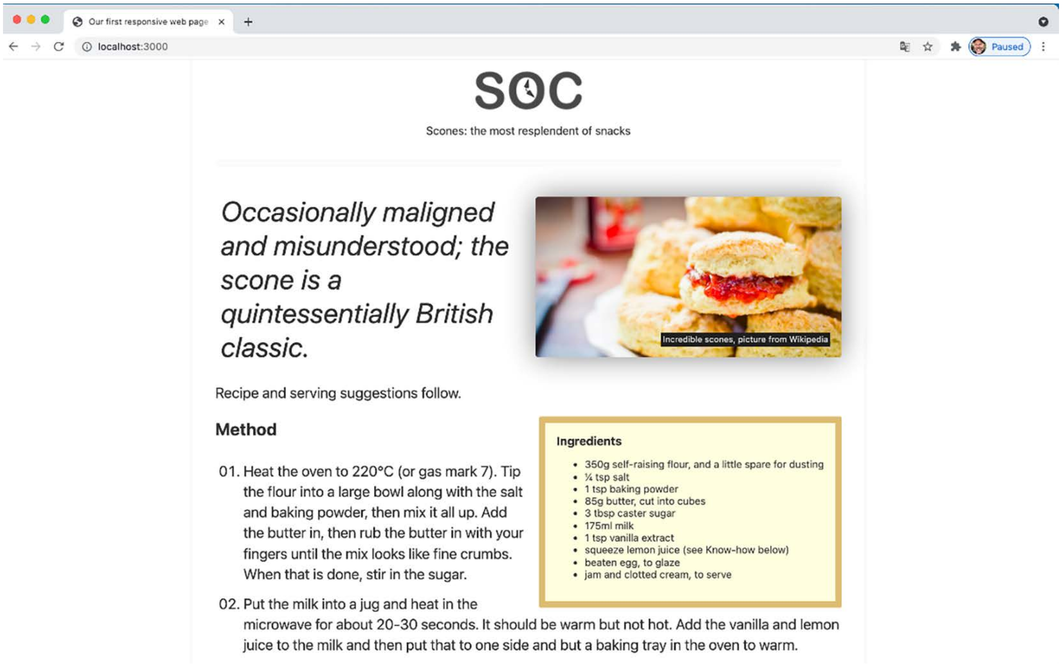
Recipe and serving suggestions follow.

Ingredients

- 350g self-raising flour, and a little spare for dusting
- ¼ tsp salt

Figure 1.7: A few more styles added and our basic page is palatable

And like this when the viewport is 800 px or wider:



Scones: the most resplendent of snacks

Occasionally maligned and misunderstood; the scone is a quintessentially British classic.

Incredible scones, picture from Wikipedia

Recipe and serving suggestions follow.

Method

01. Heat the oven to 220°C (or gas mark 7). Tip the flour into a large bowl along with the salt and baking powder, then mix it all up. Add the butter in, then rub the butter in with your fingers until the mix looks like fine crumbs. When that is done, stir in the sugar.
02. Put the milk into a jug and heat in the microwave for about 20-30 seconds. It should be warm but not hot. Add the vanilla and lemon juice to the milk and then put that to one side and but a baking tray in the oven to warm.

Ingredients

- 350g self-raising flour, and a little spare for dusting
- ¼ tsp salt
- 1 tsp baking powder
- 85g butter, cut into cubes
- 3 tbsp caster sugar
- 175ml milk
- 1 tsp vanilla extract
- squeeze lemon juice (see Know-how below)
- beaten egg, to glaze
- jam and clotted cream, to serve

Figure 1.8: The same HTML and CSS provide a different layout for larger viewports

This has been a very basic example but it has encapsulated the essential methodology of building out responsive web design.

Let's just go over the important points we have covered in this chapter and our basic example:

- Use whatever text editor you like
- Tools exist to make writing code easier but don't get hung up on what to use
- AI tools can often scaffold out markup and CSS but knowing the fundamentals will help you know whether the output is accurate and usable
- Responsive designs are made possible with a flexible layout, fluid images, and media queries
- A `meta` tag is needed in the head of your HTML so a browser knows how to render the page
- You'll want all images to be set with a max width of 100% in the CSS by default
- A breakpoint is just a point, typically a screen width, at which we use a media query to alter the design

- When you write CSS for responsive design, start with base styles that can work on any device, typically the smallest screen, and then use media queries to adapt the design for larger screens
- Scones with clotted cream and jam are really tasty

> You can find the full specifications for CSS Media Queries (Level 3) here: `http://www.w3.org/TR/css3-mediaqueries/`
>
> There is also a working draft for CSS Media Queries (Level 4) here: `http://dev.w3.org/csswg/mediaqueries-4/`

The shortcomings of our example

In this chapter, we've covered all the essential parts of building a basic responsive web page with HTML and CSS. Granted, it's not what I'd call a real looker. I'll forgive you for using words like "infantile," "lazy," and "ugly," but just do it quietly among yourselves; I have feelings, you know!

The point here is you and I both know that this basic responsive example is far from the limit of what we will likely be tasked with building day to day. Nor should it reflect the limit of what we are capable of building.

We need to cover typography, color, shadows, hover styles, semantic markup, accessibility concerns, animation, scalable graphics, forms, and so much more!

You get the picture; the truth is we have barely scratched the surface. But don't worry. That's what the rest of the book is for.

Summary

Well done – you now know about and understand the essential elements needed to create a fully responsive web page. However, as we have just discovered, there are plenty of places where things could be improved.

But that's fine. We don't just want the ability to make competent responsive web designs, we want to be able to create "best-of-breed" experiences. And as you're here, investing your time for the betterment of websites everywhere, I know you're up to the challenge.

In the next chapter, *Chapter 2*, *Writing HTML Markup*, we are going to take a deep dive into HTML5 markup. HTML is the very skeleton of any web page or application, the bedrock on which to build anything meaningful, the oxygen a website breathes, the... OK, I'm out of analogies – suffice it to say, HTML is pretty important, so let's press on.

2

Writing HTML Markup

HTML stands for **HyperText Markup Language**. Conceptually, HTML's purpose is to aid machines and systems in understanding a human language, such as English. It achieves this by wrapping content in a variety of tags, or "marking up" the content. You can have content on the web without CSS or without JavaScript, but you can't have human-friendly content without HTML as it is the language with which content for humans is displayed by a browser.

It's a common misconception that HTML is the "easy" part of authoring web pages and applications. Writing HTML is often dismissed as something anyone can do easily. My experience tells me otherwise. Also consider that for users who require the use of assistive technology to browse the web, the way you author HTML can mean the difference between a confusing, unusable mess and a meaningful, useful, and delightful experience. Writing good-quality HTML is not a specific need of responsive web design. It's far more important than that. It's a prerequisite of anything that you want to be accessible to all users of the web.

It's also true that these days, you can likely enter the content that you want marking up into an AI service and get pretty decent HTML given back to you. In time, the quality yielded from such services is only likely to improve, but like so much of using AI tools, the real benefits come when you can make a qualitative assessment of the output. Put simply, you really need to know if what it is giving you is any good.

This chapter is therefore going to be about writing HTML markup. We will be considering the "vocabulary" of HTML – its semantics. More succinctly, we will look at the way we can use the elements of HTML to describe the content we place in markup. HTML is what's known as a **living** standard. A few years back, the latest version was typically referred to as **HTML5**, a buzzword that helped identify modern web techniques and approaches.

It's the reason this book was originally named *Responsive Web Design with HTML5 and CSS3* instead of simply *Responsive Web Design with HTML and CSS*. Back in 2012, you could more easily highlight that your techniques were modern by using "HTML5" and "CSS3." As I write this in 2025, this distinction is less important. To read the living standard of HTML, head over here: `https://html.spec.whatwg.org/multipage/`.

Here are the topics we will cover in this chapter:

- Starting HTML pages correctly
- Possible economies of using HTML that you should avoid
- Sectioning, grouping, and text-level HTML elements
- Using Unicode in HTML and CSS
- Putting HTML elements to use
- WCAG accessibility conformance and WAI-ARIA for more accessible web applications
- Embedding media
- Responsive video and iframes
- Creating popups with the dialog element

HTML also provides specific tools for handling forms and user input. This set of features takes much of the burden away from more resource-heavy scripting for form validation. However, we're going to look at HTML forms separately in *Chapter 14*, *Forms*.

The basic structure of an HTML page looks like this:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Web Page Structure</title>
    </head>
    <body></body>
</html>
```

When writing HTML, you will typically be **marking up** or writing content inside a series of **tags** or **elements**. Tags are defined by the "less than" (<), and "greater than" angle brackets (>) around the tag name. The majority of elements in HTML have an opening and closing tag, with the closing tag differentiated by the forward slash before the element name.

To exemplify the opening and closing nature of HTML tags, a paragraph of text would be most suitably marked up with an opening at the beginning and a closing at the end. A few elements, like the `meta` tag in the example above, are known as **void** elements. These are not made of an opening and closing tag, with content between; instead, they are known as **self-closing** – notice the closing forward slash at the end.

There's only a limited number of self-closing, or void, elements, which are listed here: `https://html.spec.whatwg.org/multipage/syntax.html#void-elements`. Presently, the void tags are `area`, `base`, `br`, `col`, `embed`, `hr`, `img`, `input`, `link`, `meta`, `source`, `track`, and `wbr`.

HTML tags, which create elements, can also have one or more attributes applied to them. Some attributes are only applicable to certain elements. They are the method by which we add additional information to a tag. Look at the following example piece of HTML. Here, we link to a style sheet that we want to include in a page:

```
<link href="CSS/main.css" rel="stylesheet" type="text/css" />
```

There are three attributes in use here: `href`, `rel`, and `type`. For now, we don't need to concern ourselves with what they are doing. Just notice that they are delimited by being inside the opening tag with a word for the name of the attribute, an equals sign, and then the value of the attribute.

Although we are about to start our understanding of HTML by covering the head section, which is the content between the opening `<head>` and closing `</head>` tags, be aware that the lion's share of HTML authoring is done between the `<body>` and `</body>` tags.

However, as ever, the best place to start is at the beginning.

Getting the start of HTML pages right

Let's consider the opening elements of an HTML page and ensure we fully understand all the essential component parts.

Like so many things with the web, remembering the exact syntax of each thing inside the head section is not particularly important. What is important is understanding what each thing is for. I generally copy and paste the opening code each time, or have it saved in a text snippet, and I would recommend you do too. The first few lines in an HTML page should look something like this:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
```

The doctype

So, what do we actually have there? First of all, we opened our document with the HTML5 `Doctype` declaration:

```
<!DOCTYPE html>
```

If you're a fan of lowercase, then `<!doctype html>` is just as good. It makes no difference.

The HTML tag and lang attribute

After the `Doctype` declaration, we open the `HTML` tag – the first, and therefore **root** tag for our document. We also use the `lang` attribute to specify the language for the document, and then we open the `head` section:

```
<html lang="en">
    <head>
```

Specifying alternate languages

According to the W3C specifications (`https://html.spec.whatwg.org/multipage/dom.html#the-lang-and-xml:lang-attributes`), the `lang` attribute specifies the primary language for the element's contents and for any of the element's attributes that contain text. You can imagine how useful this will be to assistive technologies such as screen readers, as without the correct attribute, efficient reading of the content will be compromised. If you're not writing pages in English, you'd best specify the correct language code. For example, for Japanese, the HTML tag would be:

```
<html lang="ja">
```

For a full list of languages, take a look at `https://www.iana.org/assignments/language-subtag-registry`.

Character encoding

Finally, we specify the character encoding standard we want to be used on the document, which, in simple terms, tells the browser how to parse the information contained within. As the `meta` tag is a void element, it doesn't require a closing tag:

```
<meta charset="utf-8" />
```

Unless you have a specific reason not to, the value for the charset should always be `utf-8`, which includes nearly all the symbols used around the world.

The forgiving nature of HTML5 markup

If you're conscientious about how you write HTML markup, you'll typically use lowercase for the most part, wrap attribute values in straight quotation marks (not curly ones!), and declare a `type` for scripts and style sheets you link to. For example, perhaps you link to a style sheet like this:

```
<link href="CSS/main.css" rel="stylesheet" type="text/css" />
```

The fact is that HTML5 doesn't require such precision; it's just as happy to see this:

```
<link href=CSS/main.css rel=stylesheet >
```

Did you notice that? There's no end forward slash at the end of the self-closing void link tag, there are no quotation marks around the attribute values, and there is no type declaration. However, easy-going HTML5 doesn't care. The second example is just as valid as the first. This more lax syntax applies across the whole document, not just linked assets. For example, specify a `div` like this if you like:

```
<div id=wrapper>
```

That's perfectly valid HTML5. The same goes for inserting an image:

```
<img SRc=frontCarousel.png alt="front Carousel">
```

That's also valid HTML5. No end tag/slash, no quotes (although you still need quotes where the value has white space in, like in the `alt` tag of the example), and a mix of capitalization and lowercase characters. You can even omit things such as the opening `<head>` tag and the page still validates!

> Want a shortcut to great HTML5 code? Consider the HTML5 Boilerplate (`https://html5boilerplate.com`). It's a pre-made "best practice" HTML5 file. You can also custom-build the template to match your specific needs.

So, given the loose rules, what's the "right" way to be writing your markup?

A sensible approach to HTML markup

Personally, I like writing my markup quite strictly. That means closing tags, quoting attribute values, and adhering to a consistent letter case. One could argue that ditching some of these practices would save a few bytes of data, but that's what minification tools are for (any unnecessary characters/data could be stripped if needed). I want my markup to be as legible as possible and I would encourage others to do the same.

I’m of the opinion that clarity in authoring code should trump brevity. When writing HTML documents, therefore, you can write clean and legible code while still embracing the economies afforded by HTML5. To exemplify, for a CSS link, I’d go with the following:

```
<link href="CSS/main.css" rel="stylesheet" />
```

I’ve kept the closing forward slash at the end of the element and the quotation marks, but omitted the `type` attribute. The point to make here is that you can find a level you’re happy with yourself. HTML5 won’t shout at you, flag up your markup in front of the class, and stand you in a corner for inconsistent markup. However you want to write your markup is just fine…

No. Who am I kidding? I can’t let it go – I want you to know that if you’re writing your code without quoting attribute values and closing your tags, I am silently judging you!

> Despite HTML5’s looser syntax, it’s always worth checking whether your markup is valid. Checking that markup validates catches basic human errors such as missing or mismatched tags, missing `alt` attributes on images, incorrectly nested elements, and so on. The W3C validator was created for just this reason: `https://validator.w3.org/`.

The value of consistent approaches to markup is the reason tools such as Prettier are so popular with teams of developers; they enforce sensible and consistent markup standards, with the added benefit of automatically formatting the content for you. As with many such tools, there is some setup burden, but I can highly recommend Prettier for any project you are likely to be working on for a length of time.

Enough of me berating writers of “slacker” markup. Let’s look at some genuinely useful economies of HTML5.

All hail the mighty <a> element

The `a` tag (short for anchor tag) is arguably the most important and defining tag of HTML. The anchor tag is the tag used to link from the document a user is on to another document elsewhere on the internet, or another point in the same document.

You can read the specification for the `<a>` element here: `https://html.spec.whatwg.org/multipage/text-level-semantics.html#the-a-element`.

A welcome benefit of HTML5 is that we can wrap multiple elements in an `a` tag. In prior versions of HTML, if you wanted your markup to validate, it was necessary to wrap each element in its own `a` tag. For example, look at the following code – a heading, a paragraph, and an image all requiring separate `a` tags to link to a common destination:

```
<h2><a href="index.html">The home page</a></h2>
<p><a href="index.html">This paragraph also links to the home page</a></p>
<a href="index.html">
    <img src="home-image.png" alt="A rendering of the home page" />
</a>
```

Nowadays, we can ditch all the individual `a` tags and instead wrap the group, as demonstrated in the following code:

```
<a href="index.html">
    <h2>The home page</h2>
    <p>This paragraph also links to the home page</p>
    <img src="home-image.png" alt="A rendering of the home page" />
</a>
```

The only limitations to keep in mind with `a` tags are that, understandably, you can't wrap one `a` tag within another `a` tag or other interactive element (such as a `button`) and you can't wrap a `form` tag in an `a` tag either.

That's not to say you can't physically do it; I doubt your text editor is going to start a fight with you about it, but don't be surprised if things don't work as expected in the browser if you do!

New semantic elements in HTML5

My dictionary defines semantics as "the branch of linguistics and logic concerned with meaning." For our purposes, semantics is the process of giving our markup meaning. Why is this important?

Most websites follow fairly standard structural conventions; typical areas include a header, a footer, a sidebar, a navigation bar, and so on. As web authors, even if only to aid the visual layout we are hoping to achieve, we will often name the divs we use, to more clearly designate these areas (for example, `<div class="Header">`). However, as far as the code itself goes, any user agent parsing that content – and that includes a web browser, screen reader, search engine crawler, and so on – couldn't say for sure what the purpose of each of these `div` elements is. From their point of view, `<div class="Header">` is no more significant than `<div class="Sausages">`. HTML5 solves that problem with semantic elements to mark up our content.

For the full list of HTML5 elements, get yourself (very) comfy and point your browser here: `https://html.spec.whatwg.org/multipage/semantics.html#semantics`.

We won't cover every element here, merely those I feel are the most beneficial or interesting in day-to-day responsive web design use.

As you read through them, depending upon the kind of content you work with, some elements will seem apt, and others will seem superfluous. That should be expected, as the elements are designed to cover an enormity of use cases. As long as there are some elements that can improve the quality of your own content, and your appreciation of the utility of other elements has broadened, our time together in this chapter has been worthwhile.

After we have gone through the elements and got an understanding of their intended use, we will look at some content examples and consider how they might be best marked up. Then, to end the chapter, I'm going to set you a bigger challenge!

In terms of the HTML specification, the elements we will be looking at fall into one of three groups:

- **Sectioning elements**, for the broadest strokes in an HTML page. These are the kinds of elements to use for header, footer, and sidebar areas.
- **Grouping elements**, which are used to wrap associated elements. Think of paragraphs, blockquotes, and content of that nature.
- **Text-level semantics**, which are the elements we use to designate particulars, such as a section of bold or italic text or code.

We will now look at the most commonly used elements from each of these sections in turn. The aim here is to acquaint you with the varied vocabulary of HTML. Knowing that elements exist to convey certain meanings is a huge part of making accessible websites and applications.

First up, let's consider the various sectioning elements.

The <main> element

For a long time, HTML5 had no element to demarcate the main content of a page. It was argued that the content that wasn't inside one of the other sectioning HTML5 elements would, by negation, be the main content. Thankfully, we now have a more declarative way to group the main content: the aptly named `main` tag. Whether you're wrapping the main content of a page or the main section of a web-based application, the `main` element is what you should be grouping it all with. Here's a particularly useful line from the specification:

The main content area of a document includes content that is unique to that document and excludes content that is repeated across a set of documents such as site navigation links, copyright information, site logos and banners, and search forms (unless the document or application's main function is that of a search form).

It's also worth noting that there shouldn't be more than one `main` on each page (after all, you can't have two main pieces of content) and it shouldn't be used as a descendent child element of some of the other semantic HTML5 elements, such as `article`, `aside`, `header`, `footer`, or `nav`.

Read the official line on the `main` element at `https://html.spec.whatwg.org/multipage/grouping-content.html#the-main-element`.

The <section> element

The `section` element is used to define a generic section of a document or application. For example, you may choose to create sections around your content; one section for contact information, another section for news feeds, and so on. It's important to understand that this element isn't intended for styling purposes; if you need to wrap an element merely to style it, you should continue to use a `div` as you would have before.

When working on web-based applications, I tend to use `section` as the wrapping element for visual components. It provides a simple way to see the beginning and end of components in the markup.

You can also qualify for yourself whether you should be using a `section` element based on whether the content you are sectioning has a natural heading within it (for example, an `h1`-`h6`). If it doesn't, it's likely you'd be better off opting for a `div`.

To find out what the W3C specification says about the `section` element, go to the following URL: `https://html.spec.whatwg.org/multipage/sections.html#the-section-element`.

The <nav> element

The `nav` element is used to wrap major navigational links to other pages or parts within the same page. As it is for use in major navigational blocks, it isn't strictly intended for use in footers (although it can be) and the like, where groups of links to other pages are common.

If you usually mark up your navigational elements with an unordered list (`ul`) and a bunch of list tags (`li`), you may be better served with a `nav` and a number of nested a tags instead.

To find out what the W3C specification says about the `nav` element, go to the following URL: `https://html.spec.whatwg.org/multipage/sections.html#the-nav-element`.

The <article> element

The `article` element, alongside `section`, can easily lead to confusion. I certainly had to read and re-read the specifications of each before it sank in. Here's my re-iteration of the specification.

The `article` element is used to wrap a self-contained piece of content. When structuring a page, ask whether the content you intend to use within an `article` tag could be taken as a whole lump and pasted onto a different site and still make complete sense. Another way to think about it is would the content that you are considering wrapping in an `article` actually constitute a separate article in an RSS feed? Obvious examples of content that should be wrapped with an `article` element would be blog posts or news stories. Be aware that if nesting `article` elements, it is presumed that the nested `article` elements are principally related to the outer article.

You can read the specification for the `article` element here: `https://html.spec.whatwg.org/multipage/sections.html#the-article-element`.

The <aside> element

The `aside` element is used for content that is tangentially related to the content around it. In practical terms, I often use it for sidebars, or content in a little tip about a related subject in a blog post. It's also considered suitable for pull quotes, advertising, and groups of navigation elements. Basically, anything not directly related to the main content would work well in an aside. If it was an e-commerce site, I'd consider areas such as *Customers who bought this also bought* as prime candidates for an `aside`.

For more on what the W3C specification says about `<aside>`, visit `https://html.spec.whatwg.org/multipage/sections.html#the-aside-element`.

The <header> element

Practically, the `header` can be used for the "masthead" area of a site's header but also as an introduction to other content, such as an introduction section within an `article` element. You can use it as many times on the same page as needed. You could have a `header` inside every section on your page, for example.

> Here's what the W3C specification says about `header`: `https://html.spec.whatwg.org/multipage/sections.html#the-header-element`.

The <footer> element

Like `header`, the `footer` element doesn't take part in the outline algorithm (more on that below) so doesn't section content. Instead, it should be used to contain information about the section it sits within. It might contain links to other documents or copyright information. Like `header`, it can be used multiple times within a page if needed. For example, it could be used for the footer of a blog but also a footer within a blog post article on that same blog. However, the specification notes that contact information for the author of a blog post should instead be wrapped by an `address` element.

> The W3C specification for the `<footer>` element can be found here: `https://html.spec.whatwg.org/multipage/sections.html#the-footer-element`.

h1-h6 elements

When you write content it's typical, and preferable, to include headings for the various sections of content. Headings are marked up with levels of headings from `h1` as the highest, to `h6` as the lowest`.` A lower-number heading will have fewer ancestor sections of content. Why 6 and only 6 levels? The only answer I can find is that, originally, for technical reasons, a finite number was needed, and the use case of needing more than 6 levels of sub-sections seemed incredibly unlikely.

Using headings in your content is very important, not just visually but for users of assistive technology. As a developer, you can easily style any tag to look like a heading, and indeed style any heading to look like it isn't a heading, but styling does not help a non-sighted user understand the structure of your page, or help machines parse your code.

As a developer, it is important to understand that the use of heading elements allows a browser to build up an outline of your document, which has benefits. It makes it far easier for a non-sighted user to be able to skip to a relevant section of the content and generally navigate this page and if you are purely motivated by cold hard cash, it will also make it easier and preferable for search engines to rank your content higher. A well-structured document is far more useful to a search engine.

Some quick rules to consider with regard to headings:

- If you have any headings on a page, at least one should be `h1`.
- You shouldn't skip heading levels. If you have `h2`, and some `h4` headings, there should be a `h3` heading between them in the hierarchy. Now, this is definitely a case of "do as I say, not do as I do" as I have certainly been guilty of inadvertently skipping heading levels when structuring documents from time to time. It does not mean the document will not "work"; it just means it isn't as good as it could be. And who here wants to be sub-optimal?

Headings, sub-headings, and the hgroup element

Be aware that using `h1`-`h6` tags to mark up headings and sub-headings is discouraged. I'm talking about this kind of thing:

```
<h1>Scones:</h1>
<h2>The most resplendent of snacks</h2>
```

That would actually indicate a main heading with no actual content. In this case, you can do this, where the sub-heading is a <p> tag:

```
<h1>Scones:</h1>
<p>The most resplendent of snacks</p>
```

But even better would be to wrap it in the `hgroup` element, which is used for the very purpose of representing a heading and related content.

> The `hgroup` element was present in earlier versions of the HTML5 specification, was subsequently deprecated, and is now back again. Sometimes I wish the specification writers would make their minds up!

```
<hgroup>
    <h1>Scones:</h1>
     <p>The most resplendent of snacks</p>
</hgroup>
```

Right, we've now covered the lion's share of the HTML sectioning elements. Let's now consider grouping elements. These are the elements that are used to contain, you guessed it, groups of other elements.

The <div> element

The most ubiquitous grouping element is `div`. The `div` element is so widely used because it is opinion-less. It conveys nothing. The only implied meaning of `div` is that it groups something. Despite that, you will often see `div` wrapping nothing but a string of text.

You should only opt for `div` as a last resort. It is the element to use when you can think of nothing better.

We have more elements in HTML than ever before so, as we continue, hopefully we can get acquainted with plenty of other elements that can be more suitable for the jobs you currently use `div` for.

The <p> element

The `p` element is used to mark up a paragraph. However, don't think that means it can only be used on text 3-4 lines long. On the contrary, use it to mark up any text that cannot be better marked up with one of the other elements. For non-specific text, the `p` element is definitely a better choice than `div`. The `p` element at least conveys that text is expected inside, whereas `div` conveys that anything could be.

The <blockquote> element

A blockquote is used to mark up text that is quoted from somewhere else. You don't have to wrap the text inside with any other element, but you can. For example, knowing what we now do about the `p` tag, we can use that inside `blockquote`, too, if we wish. Here is a simple example using `blockquote`. First, an introductory section of text in a `p` tag, and then a blockquote:

```
<p>I did like Ben's book, but he kept going on about scones. For
example:</p>
<blockquote>
    All this writing about scones in our sample page and there's no image
    of the beauties! I'm going to add in an image of a scone near the top
    of the page; a sort of 'hero' image to entice users to read the page.
</blockquote>
```

There are some good examples of how and when to use blockquotes over on the HTML specification too: `https://html.spec.whatwg.org/multipage/grouping-content.html#the-blockquote-element`.

The <figure> and <figcaption> elements

The HTML specification relates that the `figure` element:

> *...can thus be used to annotate illustrations, diagrams, photos, code listings, etc.*

So, we use it as an element to call out visuals of any sort and the accompanying `figcaption` provides the means to add some text supporting the visuals. Now, it is worth pointing out here that while we should always provide text in the `alt` attribute of an `img` tag to support assistive technology or to mitigate problems if an image fails to load, it isn't a requirement to provide a `figcaption` with a `figure`. The `figcaption` is added if you want to add a visual description alongside the visuals. Here's how we could use it to revise a portion of markup from the first chapter:

```
<figure class="MoneyShot">
    <img
        class="MoneyShotImg"
        src="img/scones.jpg"
        alt="Incredible scones baked to perfection and ready to eat"
    />
    <figcaption class="ImageCaption">
        This image isn't of scones I have made, instead it's a stock photo
        from Wikipedia
    </figcaption>
</figure>
```

You can see that the `figure` element is used to wrap this little self-contained block. Inside, the `figcaption` is used to provide a caption for the parent `figure` element. It's perfect when images or code need a little caption alongside (that wouldn't be suitable in the main text of the content).

The specification for the `figure` element can be found here: `https://html.spec.whatwg.org/multipage/grouping-content.html#the-figure-element`.

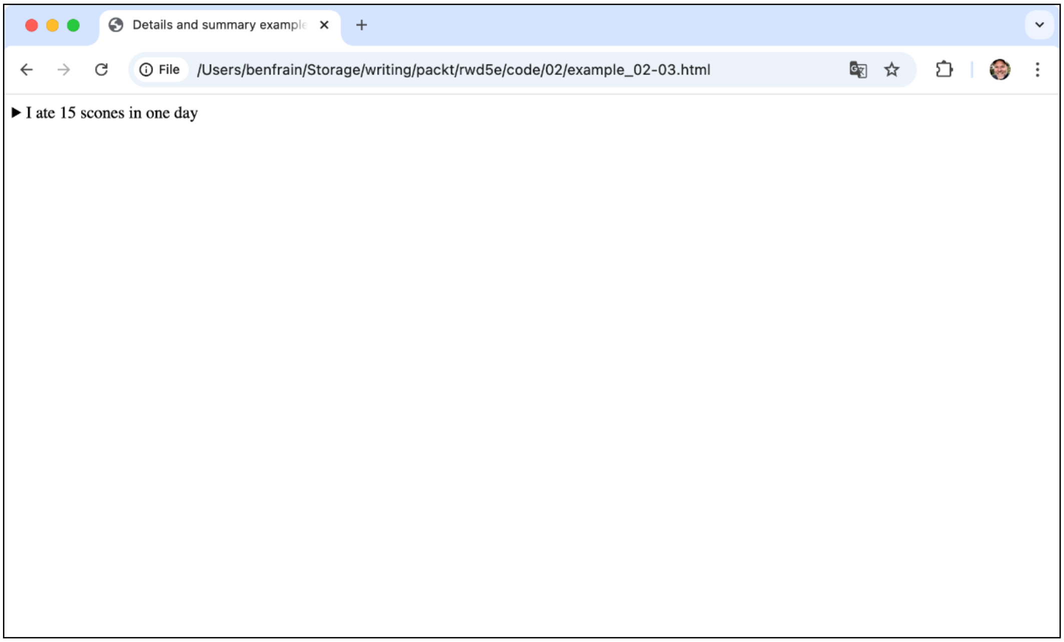
The specification for `figcaption` is here: `https://html.spec.whatwg.org/multipage/grouping-content.html#the-figcaption-element`.

The <details> and <summary> elements

How many times have you wanted to create a simple open and close "widget" on your page? A piece of summary text that, when clicked, opens a panel with additional information. Modern HTML facilitates this pattern with the `details` and `summary` elements. Because it is native HTML, you get the added accessibility benefits of being able to press *Tab* to select the element, and the spacebar to open it, all "for free." Consider this markup (you can open `example_02-03.html` from this chapter's code to play with it for yourself):

```
<details>
    <summary>I ate 15 scones in one day</summary>
    <p>
        Of course I didn't. It would probably kill me if I did. What a
        way to go. Mmmmmm, scones!
    </p>
</details>
```

Opening this in Chrome, with no added styling, shows only the summary text by default:

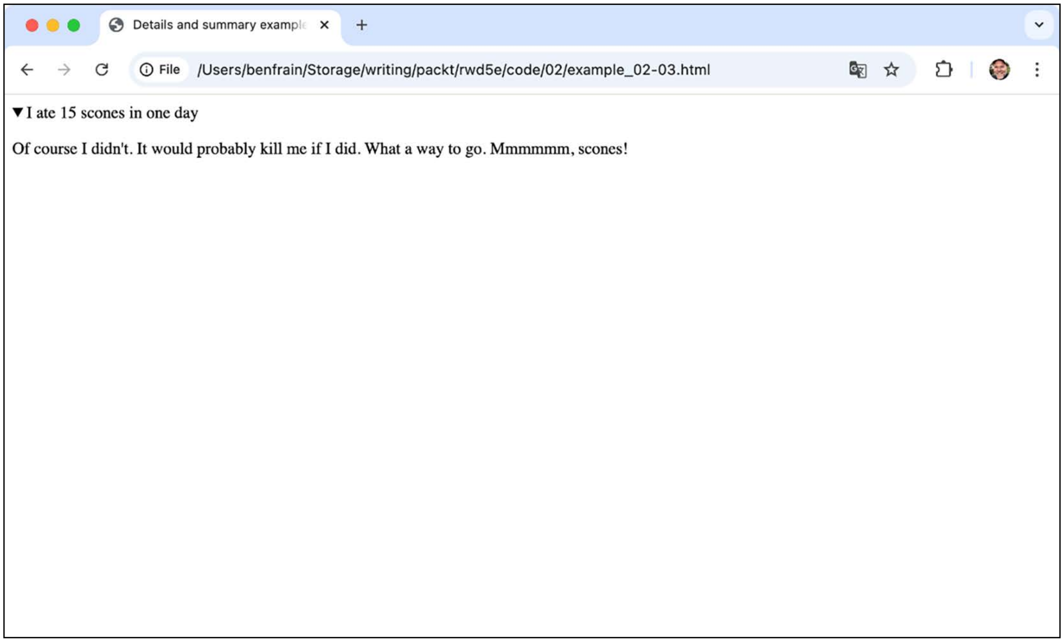

Details and summary example
File
/Users/benfrain/Storage/writing/packt/rwd5e/code/02/example_02-03.html

▶ I ate 15 scones in one day

Figure 2.1: details and summary attempt to solve a common problem but their implementation is limited

Clicking anywhere on the `summary` text opens the panel. Clicking it again toggles it shut. If you want the panel open by default, you can add the `open` attribute to the `details` element:

```
<details open>
    <summary>I ate 15 scones in one day</summary>
    <p>
        Of course I didn't. It would probably kill me if I did. What a way
        to go. Mmmmmm, scones!
    </p>
</details>
```


Details and summary example
File
/Users/benfrain/Storage/writing/packt/rwd5e/code/02/example_02-03.html

▼I ate 15 scones in one day

Of course I didn't. It would probably kill me if I did. What a way to go. Mmmmmm, scones!

Figure 2.2: By adding the open attribute, the content is shown by default

Browsers typically add some default styling to indicate the panel can be opened. Changing the style is frustrating though as there is no cross-browser consensus on how to do parts of it. Different browsers have their own implementations of how we can deal with any visual marker for `details`. In Safari, you need to use a proprietary pseudo-class (note the `-webkit-` prefix) to remove it entirely:

```
summary::-webkit-details-marker {
    display: none;
}
```

You can, of course, use that same selector to style the marker differently but not change the symbol. Although, if need be, you could hide the default marker and add your own as a background image on a `::before` or `::after` pseudo-element, for example.

To hide the marker in Chrome and Firefox, you can do this:

```
summary::marker {
    content: none;
}
```

In addition, you can also change the icon used in Chrome by using the `list-style-type` property. So, if you wanted to use an alternative character for the open/closed styles, you could do this.

```
summary {
    list-style-type: "↓ ";
}
[open] summary {
    list-style-type: "↑ ";
}
```

If the differences in styling weren't annoying enough, currently, there is no cross-browser way of animating the open and close. It works in Chromium browsers such as Chrome and Edge but Safari does not support it at the time of writing. There is quite a bit of modern CSS in the following code, which we cover in later chapters. For now, you can just appreciate it is possible to animate the open and close in Chromium browsers by adding the following. You can play about with this by commenting and uncommenting the various sections in the style section of the example code.

```
:root {
    interpolate-size: allow-keywords;
}
details {
    overflow: hidden;
}
details::details-content {
    block-size: 0;
    transition:
        block-size 0.5s,
        content-visibility 0.5s;
    transition-behavior: allow-discrete;
}
```

```
details[open]::details-content {
    block-size: auto;
}
```

The final thing to tell you is that, nowadays, you can also set details and summary to behave like an accordion, and it is very simple to do so.

Create an accordion effect with multiple linked details elements

There may be times when you want two or more `details` elements to work as an accordion, whereby if you open one, the other closes. I'm not a huge fan of this pattern in general, as if a user has opted to show content, why then hide it from them when they click something else? But there may be occasions when this is needed. Creating this association is as simple as adding a `name` attribute with the same value to each `details` element you want to be a part of the accordion. If you uncomment the relevant section of code in the `example_02-03.html` file, you will see a couple of related `details`. Try opening one, and then opening the other. I've also added a little styling to these, and you can see that the styles and pseudo-selectors don't match across browsers currently, but this will aid you in seeing the pattern in action. None of the additional styling is actually required to get the accordion functionally.

The <address> element

The `address` element is to be used explicitly for marking up contact information for its nearest `article` or `body` ancestor. Keep in mind that it isn't to be used for postal addresses and the like (unless they are indeed the contact addresses for the content in question). Instead, postal addresses and other arbitrary contact information should be wrapped in good ol' `p` tags. I'm not a fan of the `address` element. In my experience, it would be far more useful to mark up a physical address with this element. However, hopefully it makes more sense to you.

For more on what the W3C specification says about the `<address>` element, go to the following page: `https://html.spec.whatwg.org/multipage/sections.html#the-address-element`.

We have now covered the majority of the sectioning elements of HTML, so we will move on to the text-level elements. These are the elements used to mark up individual words, letters, and symbols to provide explicit meaning as to their intent.

HTML text-level semantics

Before HTML5, text-level semantic elements were referred to in the specifications as `inline` elements. Therefore, if you are familiar with that description, be aware that we are talking about the same thing here.

The section of the HTML specification that details text-level semantics can be found here: `https://html.spec.whatwg.org/multipage/text-level-semantics.html#text-level-semantics`.

Let's take a look at the most common and useful text-level elements.

The <span> element

A `span` element is the text-level equivalent of `div`. It is unopinionated and is the perfect element to reach for when you merely want to wrap text in an element for styling purposes.

The <b> element

Historically, visuals were defined in the markup and the `b` element meant "make this bold" (`https://www.w3.org/TR/html4/present/graphics.html#edef-B`). However, the specification now describes the `b` element like this:

> *The b element represents a span of text to which attention is being drawn for utilitarian purposes without conveying any extra importance and with no implication of an alternate voice or mood, such as key words in a document abstract, product names in a review, actionable words in interactive text-driven software, or an article lede.*

Although no specific meaning is now attached to it, as it's text-level, it's not intended to be used to surround large groups of markup; use `div` for that.

You should also be aware that because it was historically used to bold text, you'll typically have to reset the font weight in CSS if you want content within a `b` tag to not appear bold.

Here's an example:

```css
b {
    font-weight: normal;
}
```

The <strong> element

If you do want to emphasize something for strength, urgency, or importance, `strong` is the element for you. Here is how the specification defines these use cases:

Importance: The strong element can be used in a heading, caption, or paragraph to distinguish the part that really matters from other parts that might be more detailed, more jovial, or merely boilerplate. Seriousness: The strong element can be used to mark up a warning or caution notice. Urgency: The strong element can be used to denote content that the user needs to see sooner than other parts of the document.

> You can read the full specification for `strong` here: `https://html.spec.whatwg.org/multipage/text-level-semantics.html#the-strong-element`.

The <em> element

I'll admit, in the past, I've often used `em` merely as a styling hook, to provide italic text when I wanted it. I need to mend my ways as the HTML specification tells us:

> *The em element represents stress emphasis of its contents.*

Therefore, unless you actually want the enclosed contents to be emphasized, consider using a `b` tag or, where relevant, an `i` or `span` tag instead.

The <i> element

The HTML5 specification describes `i` as follows:

> *A span of text in an alternate voice or mood, or otherwise offset from the normal prose in a manner indicating a different quality of text.*

Suffice it to say, it's not to be used to merely italicize something. For example, we could use it to mark up the odd name in this line of text:

```html
<p>
    However, discussion on frameset is now frustraneous as it's now
    gone the way of the <i>Raphus cucullatus</i>.
</p>
```

Or, perhaps if you were marking up a button in a food ordering web application, you might do this:

```html
<button type="button">French Fries <i>No Salt Added</i></button>
```

There are plenty of other text-level semantic tags in HTML; for a full rundown, take a look at the relevant section of the specification at the following URL: `https://html.spec.whatwg.org/multipage/text-level-semantics.html#text-level-semantics`.

Obsolete HTML features

If you have been writing HTML for a few years, you might be surprised to learn there are parts of HTML that are now considered "obsolete." It's important to be aware that there are two camps of obsolete features in HTML: **conforming** and **non-conforming**. Conforming features will still work, but will generate warnings in validators. Realistically, avoid them if you can, but they aren't going to make the sky fall down if you do use them. Non-conforming features may still render in certain browsers but there is an equal chance they won't.

In terms of obsolete and non-conforming features, there is quite a raft. I'll confess that many I have never used. Some I've never even seen! It's possible you may experience a similar reaction. However, if you're curious, you can find the full list of obsolete and non-conforming features at `https://html.spec.whatwg.org/multipage/obsolete.html`. Notable obsolete and non-conforming features that you may occasionally encounter are strike, center, font, acronym, frame, and frameset.

How to insert Unicode characters in HTML and CSS

Unicode is the use of a number to represent a symbol, and the use of these numbers means that any character, from any language, can be effectively represented in a document of any other language. In my experience, I encounter Unicode when I need to insert particular symbols. Think of things like TM for trademark, or the copyright symbol. The copyright symbol is represented in Unicode in a number of ways. Most commonly for HTML we are used to seeing it represented as a decimal number, `169`, or as a hexadecimal number, `00A9`.

For HTML, there are also aliases for these numbers known as named entities. These allow you to write something a little more human-friendly. For example, in HTML, if I want to use the copyright symbol, I can write `&copy;` and the trademark symbol would be `&trade;`. In both instances, you can see that the syntax is the ampersand, then a name, and then a semi-colon. So, in the case of the copyright symbol, in HTML, all these render the same:

```
<p>&copy; copyright 2025</p>
<p>&#0169; copyright 2025</p>
<p>&#x00A9; copyright 2025</p>
```

To confuse matters, the format for entering a Unicode symbol in CSS is slightly different. Here, you need to use a Unicode escape sequence where the "escape" for the sequence is the slash, and a space is used afterward as a "terminator" to signify the end of the sequence. In the following example code, there are two spaces after the number. One is the terminator and the other is the actual space I want to see.

```
body {
  &::after {
    content: "\00A9  copyright 2025";
  }
}
```

You can see that we are using the same hexadecimal number as before but using a different syntax to delimit it.

In each of these cases, it is also worth noting that the leading zeroes are superfluous.

> You can find a list of Unicode symbols on Wikipedia here: `https://en.wikipedia.org/wiki/List_of_XML_and_HTML_character_entity_references`. Sometimes you have already got the symbol and need to find the correct representation of it in Unicode. In that case, this page lets you paste the symbol in and have it convert it to the various codes you may need: `https://r12a.github.io/app-conversion/`.

Putting HTML elements to use

It's time to practice using some of the elements we have just looked at. Let's revisit the example from *Chapter 1*. If we compare the markup below to the original markup in *Chapter 1* (remember, you can download all the examples from the `https://rwd.education` website), you can see where the new elements we've looked at have been employed here:

```
<article>
    <header class="Header">
        <a href="/" class="LogoWrapper">
            <img src="img/SOC-Logo.png" alt="Scone O'Clock logo"
        /></a>
        <h1 class="Strap">Scones: the most resplendent of snacks</h1>
    </header>
    <section class="IntroWrapper">
        <p class="IntroText">
```

```
            Occasionally maligned and misunderstood; the scone is a
            quintessentially British classic.
        </p>
        <figure class="MoneyShot">
            <img
                class="MoneyShotImg"
                src="img/scones.jpg"
                alt="Incredible scones"
            />
            <figcaption class="ImageCaption">
                Incredible scones, picture from Wikipedia
            </figcaption>
        </figure>
    </section>
    <p>Recipe and serving suggestions follow.</p>
    <section class="Ingredients">
        <h3 class="SubHeader">Ingredients</h3>
    </section>
    <section class="HowToMake">
<h3 class="SubHeader">Method</h3></section>
    <footer>
        Made for the book,
        <a href="https://rwd.education">
            'Responsive web design with HTML5 and CSS'</a>
        by
        <address><a href="https://benfrain">Ben Frain</a></address>
    </footer>
</article>
```

I've removed a good portion of the inner content so we can concentrate on the structure. Hopefully, you will agree that it's easy to discern different sections of markup from one another. However, at this point, I'd also like to offer some pragmatic advice; it isn't the end of the world if you don't always pick the correct element for every single given situation.

For example, whether or not I used `section` or `div` in the preceding example is of little real consequence. If we use `em` when we should actually be using `i`, I certainly don't feel it's a crime against humanity; the folks at the W3C won't hunt you down and feather and tar you for making the wrong choice.

Just apply a little common sense. That said, if you can use elements such as `header` and `footer` when relevant, there are inherent accessibility benefits in doing so. Any effort you make to better mark up your content is certainly better than using nothing but divs!

WCAG accessibility conformance and WAI-ARIA for more accessible web applications

I'll repeat myself here in case you have jumped to this section from elsewhere. If you only do one thing in the pursuit of making your web pages more accessible, it should be to use the correct elements to mark up the content of your pages and applications. It is the simplest thing you can do to make your work accessible to all and well-formed markup will also rank better in search engines.

Accessibility is a consideration in many things "frontend," whether that is font sizes, color contrast, or the correct marking up of content. We will cover accessibility as it relates to those topics as we get to them in future chapters. However, at this point, I want to take a moment to explain, albeit briefly, the role of WCAG and WAI-ARIA.

Since writing the first edition of this book in 2011/2012, it has been warming to see the W3C and web browsers make strides, with both education and functionality, in making it easier for authors to make the necessary adjustments to code to make web pages more accessible to all users.

If you want an introduction to the topic, I cannot (so won't) do a better job of explaining the topic than the Accessibility Fundamentals Overview page: `https://www.w3.org/WAI/fundamentals/`. Here there are a number of videos, free courses, and articles to get you up to speed. However, to set the scene a little, I do want to relate the differences and intentions of the WCAG and WAI-ARIA.

Web Content Accessibility Guidelines (WCAG)

The Web Content Accessibility Guidelines (WCAG) are a recipe of sorts for web developers that exists to provide:

> *a single shared standard for web content accessibility that meets the needs of individuals, organizations, and governments internationally*

When it comes to more pedestrian web pages, as opposed to single-page web applications and the like, which we will come to shortly, it makes sense to concentrate on the WCAG guidelines.

These guidelines offer a number of, mostly common-sense, guidelines for how to ensure your web content is accessible. Each recommendation is rated by a conformance level: A, AA, or AAA, and each of these levels has a distinct set of acceptance criteria.

For more on these conformance levels, look here: `https://www.w3.org/TR/UNDERSTANDING-WCAG20/conformance.html#uc-levels-head`.

You'll probably find that you are already adhering to many of the guidelines, such as providing alternative text for images, for example. However, you can get a brief rundown of the guidelines here: `https://www.w3.org/WAI/standards-guidelines/wcag/glance/` and then build your own custom quick reference list of checks here: `https://www.w3.org/WAI/WCAG21/quickref/`.

I'd encourage everyone to spend an hour or two looking down the list. Many of the guidelines are simple to implement and offer real benefits to users.

WAI-ARIA

The aim of **Web Accessibility Initiative – Accessible Rich Internet Applications** (**WAI-ARIA**) is principally to solve the problem of making dynamic content on a web page accessible. It provides a means of describing roles, states, and properties for custom widgets, such as dynamic sections in web applications, so that they are recognizable and usable by assistive technology users. For example, if an onscreen widget displays a constantly updating stock price, how would a blind user accessing the page know that? WAI-ARIA attempts to solve this problem.

How to implement ARIA fully in something such as a web application is outside the scope of this book. However, if that's the kind of project you are building, head over to `https://www.w3.org/WAI/standards-guidelines/aria/` for more information.

Instead, for a great overview of getting to grips with ARIA, consider reading the W3C *Using Aria* guide: `https://www.w3.org/TR/using-aria/`.

But at the risk of sounding like a skipping record (if you are under 30 years old, ask a 50-year-old), the easiest thing you can do to aid assistive technologies is to use the correct elements where possible. A `header` element is going to be far more useful than `<div class="Header">`. Similarly, if you have a button on your page, use the `button` element, rather than a `span` or other element styled to look like a button.

Taking ARIA further

If you want to explore a solid set of accessible design patterns, the W3C has a published set you can peruse here: `https://www.w3.org/WAI/ARIA/apg/example-index/#examples_by_props_label`.

Test your designs for free with **NonVisual Desktop Access (NVDA)**. If you develop on the Windows platform and you'd like to test your ARIA-enhanced designs on a screen reader, you can do so for free with NVDA. You can get it at the following URL: `https://www.nvaccess.org`. Google now also ships the free Accessibility Developer Tools for the Chrome browser (available cross-platform) – well worth checking out. There's also a growing number of tools that help quickly test your own designs against things such as color blindness; for example, deque make their Axe DevTools browser extension available: `https://www.deque.com/axe/devtools/`.

For now, hopefully, this brief introduction to WAI-ARIA and WCAG has given you a little background information on supporting assistive technology. Perhaps adding assistive technology support to your next HTML project will be easier than you think. As a final resource for all things accessibility, there are handy links and advice galore on the A11Y project home page at `https://a11yproject.com/`.

Embedding media in HTML5

Before HTML5, adding video and audio into markup was a bit of a pain. Now, it's easy.

Here's a "simple as can be" example of how to link to a video file on your page:

```
<video src="myVideo.mp4"></video>
```

HTML allows a single `video` tag (or `audio` tag for audio) to do all the heavy lifting. It's also possible to insert text between the opening and closing tags to inform users when there is a problem. There are also additional attributes you'd ordinarily want to add, such as the height and width. Let's add these in:

```
<video src="myVideo.mp4" width="640" height="480">
    If you're reading this either the video didn't load or your browser is
    waaaayyyyyy old!
</video>
```

Now, if we add the preceding code snippet to our page and look at it in some browsers, it will appear, but there will be no controls for playback. To ensure we show the default playback controls, we need to add the `controls` attribute. For the sake of illustration, we could also add the `autoplay` attribute.

These days, you will also usually need to add the `muted` attribute if you want your video to autoplay. Everyone detested videos automatically blaring out audio into a crowded office (if you want to listen to a little Rick Astley at lunch, who am I to judge) so much that most browsers won't autoplay without the `muted` attribute present.

Here's an example with the `controls` and `autoplay` attributes added:

```
<video src="myVideo.mp4" width="640" height="480" controls autoplay>
    If you're reading this either the video didn't load or your browser is
    waaaayyyyyy old!
</video>
```

The result of the preceding code snippet is shown in the following screenshot:

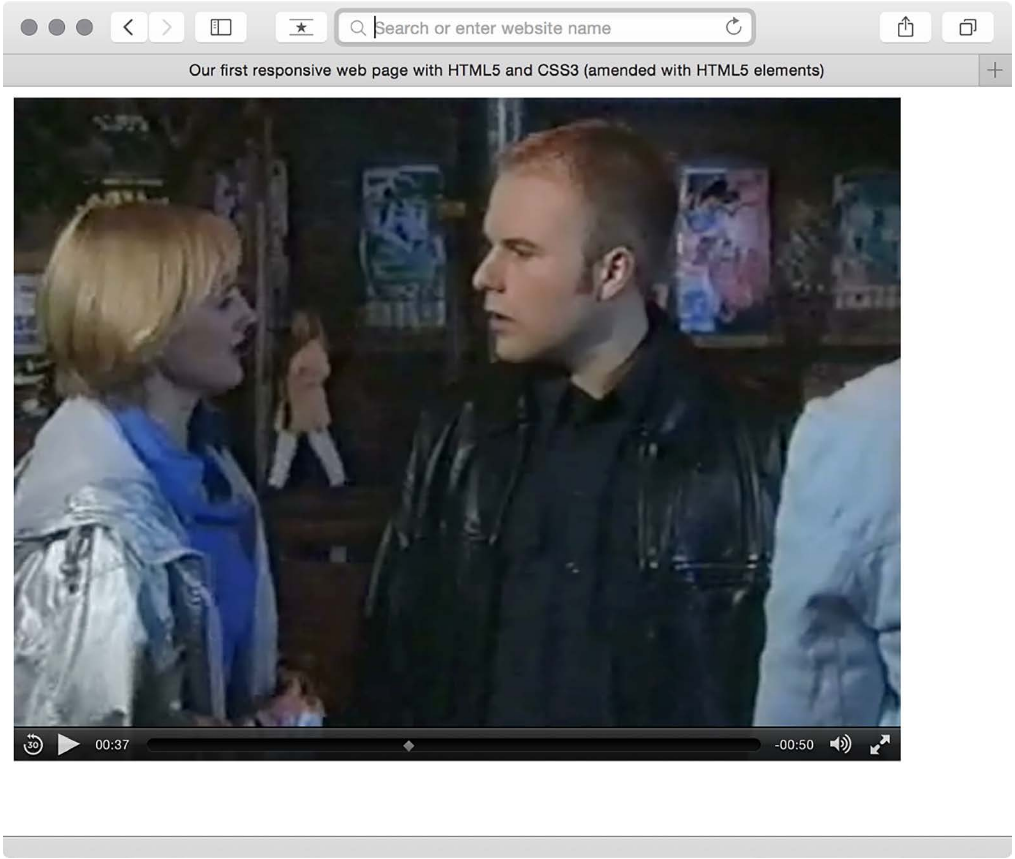

Search or enter website name
Our first responsive web page with HTML5 and CSS3 (amended with HTML5 elements)
00:37
-00:50

Figure 2.3: With minimal code, we have inserted a video into our page

Further attributes include `preload` to control the preloading of media, `loop` to repeat the video, and `poster` to define a poster frame for the video – the image that shows while a video is loading. To use an attribute, simply add it to the tag. Here's an example that includes all these attributes:

```
<video
    src="myVideo.mp4"
    width="640"
    height="480"
    controls
    autoplay
    muted
    preload="auto"
    loop
    poster="myVideoPoster.png"
>
    If you're reading this either the video didn't load or your browser is
    waaaayyyyyy old!
</video>
```

Providing alternate media sources

The `source` tag enables us to provide alternate sources for the media. For example, alongside providing an MP4 version of the video, if we wanted to provide support for a new format, we could easily do so. Furthermore, if the user didn't have any suitable playback technology in the browser, we could provide download links to the files themselves. Here's an example:

```
<video
    width="640"
    height="480"
    controls
    preload="auto"
    loop
    poster="myVideoPoster.png"
>
    <source src="video/myVideo.sp8" type="video/super8" />
    <source src="video/myVideo.mp4" type="video/mp4" />
    <p><b>Download Video:</b>
    MP4 Format: <a href="myVideo.mp4">"MP4"</a></p>
</video>
```

In that case, we first specified a source of a made-up video format called `super8`. The browser goes top to bottom deciding what to play, so if it doesn't support `super8`, it moves on to the next source – `mp4` in this case. And on and on it goes, moving down to the download link if it can't reconcile any of the sources listed. The `type` attribute tells the browser the MIME type of the resource, which is the type, then the subtype separated by a forward slash. If you fail to specify this, the browser will fetch the content and try and play it anyway. But if you know the MIME type, you should add it in the `type` attribute. The prior code example and the sample video file (which, incidentally, is me appearing in the UK TV soap opera *Coronation Street* back when I had hair and hopes of starring alongside De Niro) in MP4 format are in the *02* code folder.

Audio and video tags work almost identically

The `audio` tag works on the same principles as the `video` tag, with the same attributes (excluding `width`, `height`, and `poster`). The main difference between the two is the fact that `audio` has no playback area for visible content.

Responsive HTML5 video and iframes

The only problem with our lovely HTML video implementation is that it's not responsive. That's right, an example in a responsive web design with HTML5 and CSS book that doesn't "respond." Thankfully, for HTML-embedded video, the fix is easy. Simply remove any height and width attributes in the markup (for example, remove `width="640" height="480"`) and add the following in the CSS:

```
video {
    max-width: 100%;
    height: auto;
}
```

However, while that works fine for files that we might be hosting locally, it doesn't solve the problem of videos embedded within an `iframe` (take a bow YouTube, Vimeo, et al.). The following code will add a film trailer for *Midnight Run* from YouTube:

```
<iframe
    width="960"
    height="720"
    src="https://www.youtube.com/watch?v=B1_N28DA3gY"
    frameborder="0"
    allowfullscreen
></iframe>
```

However, if you add that to a page as is, even if adding that earlier CSS rule, if the viewport is less than 960 px wide, things will start to get clipped.

How to maintain the aspect ratio

In modern browsers, you can set the aspect ratio of something like an `iframe` with the `aspect-ratio` CSS property. For an `iframe` we want to keep in a 16:9 ratio, such as a YouTube video, we could add a suitable class and do this:

```
.iframe-16-9 {
    aspect-ratio: 16/9;
    max-width: 100%;
}
```

You can see this in action for yourself in `example_02-04.html` of the chapter code. Of course, the `aspect-ratio` property is not limited to iframes – you can use it on any box in CSS that you want to maintain certain proportions with.

The loading attribute

If you find yourself adding a lot of images or `iframe` elements to your page, consider the `loading` attribute. Setting this attribute to `lazy` means the browser will only go and fetch the item if the user scrolls it into view. This is really useful for any media you know will certainly not be visible when the page first displays, meaning the cost of loading that item is saved until it is actually needed. It's literally as simple as adding it like this:

```
<img src="home-image.png" alt="A rendering of the home page"
    loading="lazy" />
```

Or, if you were adding something like a YouTube video embed, you could make the same economy like this:

```
<iframe
    loading="lazy"
    width="560"
    height="315"
    src="https://www.youtube.com/embed/NkfMBI1tVwY"
    title="YouTube video player"
    frameborder="0"
    allow="accelerometer; autoplay; clipboard-write; encrypted-media;
    gyroscope; picture-in-picture"
    allowfullscreen
></iframe>
```

In case you were wondering, the opposite of `lazy` in this context is `eager`, which is the default if you don't add the attribute.

The <dialog> element

You know what a dialog is, even if you know it by a different name. It's the popup you get on a site asking you to subscribe to a newsletter or log in to a service. Coding this kind of thing has always been problematic. Worse still, if you didn't think they were problematic it is likely because you didn't realize there were accessibility problems with your implementation. It's likely users using only a keyboard could struggle to get focus on the dialog or be unable to leave it.

Thankfully, we now have a specific element to handle these use cases. In the code for this chapter, you will want to consider `example_02-05.html`.

Inside that HTML file is also the relevant CSS and JavaScript to produce a simple but fully working dialog.

When the modal dialog is opened, it automatically gets centered and focused, and the backdrop of the dialog — which is automatically inserted — covers everything below, preventing any of the underlying content from being interacted with. You can dismiss the dialog with the button inside it, or a keyboard press of the *Esc* key; the keyboard support is functionality you get for free!

The `dialog` is a wonderful addition to the web platform.

Here is the relevant HTML from our example:

```
        <button id="launchDialog">Where is the dialog?</button>
        <p id="formResult"></p>
        <dialog id="dialogEle">
            <form method="dialog">
                <h1>How about this? A native Dialog.</h1>
                <p>So, only thing left to do is dismiss me.</p>
                <button value="Dismissed!">Dismiss Dialog</button>
            </form>
        </dialog>
```

It's worth noting here that I have included the content of the `dialog` in a `form`. That isn't necessary, but doing so allows you to use the special `method="dialog"` attribute, which "returns" the value of the `button` doing the submitting of the form – more in-built functionality that can prove useful!

Also worth noting is that when you click the button to open a dialog, it automatically gets an open attribute, which you can see in the dev tools. This can be used as a styling hook. I've added that in the styles when animating the modal `backdrop` when it first appears, although it isn't actually necessary in this instance.

Let's take a look at the styles:

```
dialog {
    border-radius: 10px;
}
dialog::backdrop {
    animation: fade 0.5s ease forwards;
}
button {
    height: 40px;
    padding: 5px 10px;
}
button:focus {
    outline: 2px solid #f90;
}
@keyframes fade {
    0% {
        background-color: transparent;
    }
    100% {
        background-color: rgb(0 57 24 / 0.6);
    }
}
```

None of the styles are necessary for the dialog to function, but I do always like to add a stronger `:focus` style than the default for better accessibility. However, I thought it would also be interesting to bring the `::backdrop` pseudo-element to your attention, as that is how you can style the area behind the modal dialog.

If any of the syntaxes or styles in there are unfamiliar to you, don't worry. We cover all of those in detail later in the book. All that is necessary here is to understand that you can style the dialog and backdrop however you see fit.

This book is deliberately light on JavaScript, but we do need to consider some now to understand how the dialog functions:

```
        const dialogEle = document.getElementById("dialogEle");
        const launchBtn = document.getElementById("launchDialog");
        const formResult = document.getElementById("formResult");
        launchBtn.addEventListener("click", () => dialogEle.showModal());
        dialogEle.addEventListener("close", () => {
            formResult.textContent = dialogEle.returnValue;
        });
```

Besides creating `const` references to the dialog itself, the button that opens it, and a `p` tag to display the value of the button that can be clicked inside the dialog, we have two event "listeners." The first opens the dialog when the `launchBtn` button is clicked with the `showModal()` method, and the second listens for the `close` event. The `close` event is unique to `dialog` and gets triggered when submitting a form inside a `dialog` that has a method of `dialog`, or if the user exits the form with, say, a press of *Esc* on their keyboard.

> The default `type` of a `button` is `submit`, so in the preceding example, where we want to submit the form inside the dialog, there is no need to declare it. However, imagine you are building a form, as we will in detail in *Chapter 14*, *Forms*, and you have a button in your form for some other purpose; you don't want the form to be submitted when the user clicks that, rather than the correct submit button. As such, I recommend being in the habit of always adding `type="button"` whenever you use the `button` element.

In our example, we are using the `returnValue` of the `dialog` to set the text of our `p` tag. That `returnValue` is set to the `value` attribute of the button that submits the form in the dialog. Not essential, but potentially useful depending on your use case.

It's also possible to show a dialog in a non-modal manner. To do that, instead of using the `showModal()` method, use the `show()` method. In this situation, no `backdrop` is inserted and the `dialog` is inserted using absolute positioning by default.

You now know how to create, control, and style a fully working dialog.

> You can read the specification for the `dialog` element here: `https://html.spec.whatwg.org/multipage/interactive-elements.html#the-dialog-element`.

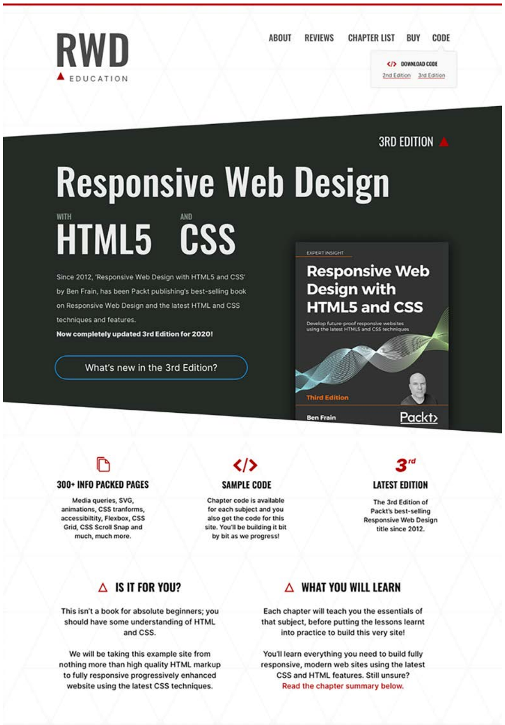
Summary

We've covered a lot in this chapter – everything from the basics of creating a page that validates as HTML5 through to embedding rich media (video and audio) into our markup and ensuring it behaves responsively. Although not specific to responsive designs, we've also covered how we can write semantically rich and meaningful code and considered how we might ensure pages are meaningful and usable for users who are relying on assistive technology.

You should now have a decent understanding of the wide and varied vocabulary of HTML elements at your disposal.

An exercise

In this chapter, we covered a raft of HTML elements. We didn't cover them all, but we certainly covered all the elements you are likely to need day to day. I think you're ready to try a little exercise to see how much you have understood. The following is a screenshot of the site design used to create the original `rwd.education` website for this book. The design has been tweaked a little over the years but it is structurally very similar to how it is today.


RWD
EDUCATION
ABOUT
REVIEWS
CHAPTER LIST
BUY
CODE
</> DOWNLOAD CODE
2nd Edition
3rd Edition
3RD EDITION
Responsive Web Design
WITH
HTML5
AND
CSS
What's new in the 3rd Edition?
EXPERT INSIGHT
Responsive Web Design with HTML5 and CSS
Develop future-proof responsive websites using the latest HTML5 and CSS techniques
Third Edition
Ben Frain
Packt>
300+ INFO PACKED PAGES
SAMPLE CODE
3rd
LATEST EDITION
IS IT FOR YOU?
WHAT YOU WILL LEARN

Since 2012, 'Responsive Web Design with HTML5 and CSS' by Ben Frain, has been Packt publishing's best-selling book on Responsive Web Design and the latest HTML and CSS techniques and features.

Now completely updated 3rd Edition for 2020!

Media queries, SVG, animations, CSS tranforms, accessibiltity, Flexbox, CSS Grid, CSS Scroll Snap and much, much more.

Chapter code is available for each subject and you also get the code for this site. You'll be building it bit by bit as we progress!

The 3rd Edition of Packt's best-selling Responsive Web Design title since 2012.

This isn't a book for absolute beginners; you should have some understanding of HTML and CSS.

We will be taking this example site from nothing more than high quality HTML markup to fully responsive progressively enhanced website using the latest CSS techniques.

Each chapter will teach you the essentials of that subject, before putting the lessons learnt into practice to build this very site!

You'll learn everything you need to build fully responsive, modern web sites using the latest CSS and HTML features. Still unsure? Read the chapter summary below.

Figure 2.4: The design for the website made for this book

If you have a Mac and you'd rather look at the original Sketch file, it's included in the download code as `RWD3e_design.sketch`.

Take a look at the design, and try and create an HTML page for it. Consider the head section and what you need in there. Think about the language the content is in and any meta tags you might need. Then, think about the visuals themselves. What might be the best elements to use to create the navigation section? Or each of those little sections below the book image? And what about that **DOWNLOAD CODE** box; any ideas on how you might mark that up?

You can see the choices I made by visiting the live site at `https://rwd.education`, but do yourself a favor and don't peek until you've tried it for yourself!

3

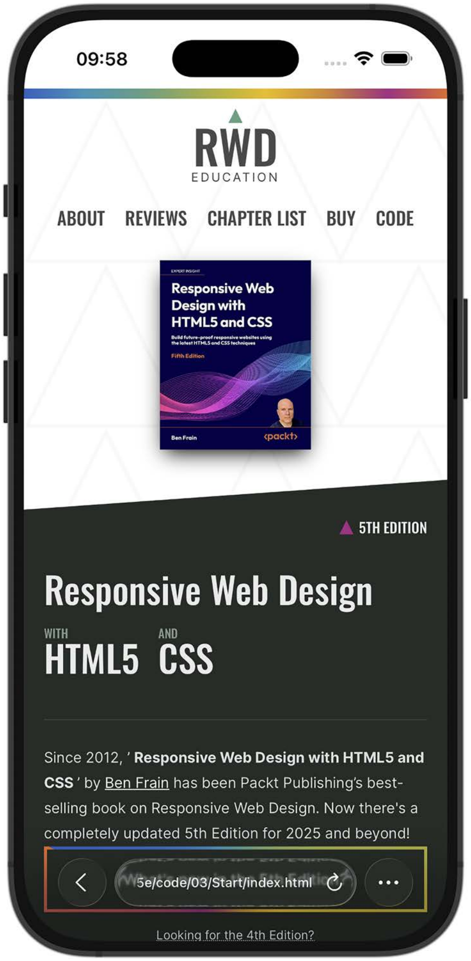
Media and Container Queries

This chapter will look in detail at CSS media and container queries, hopefully providing all that's needed to fully understand their capability and syntax.

The distinction between the two is straightforward; the former acts upon the entire viewport, the latter a container element you designate. Of the two, media queries are far more widely used and supported, and arguably needed. We will look at those in detail first.

We ended the last chapter with the initial markup for the `rwd.education` website. It wasn't necessarily the final markup, but it was solid enough for us to start working with. For the sake of this chapter, I have taken that markup and added some corresponding basic "mobile" styles.

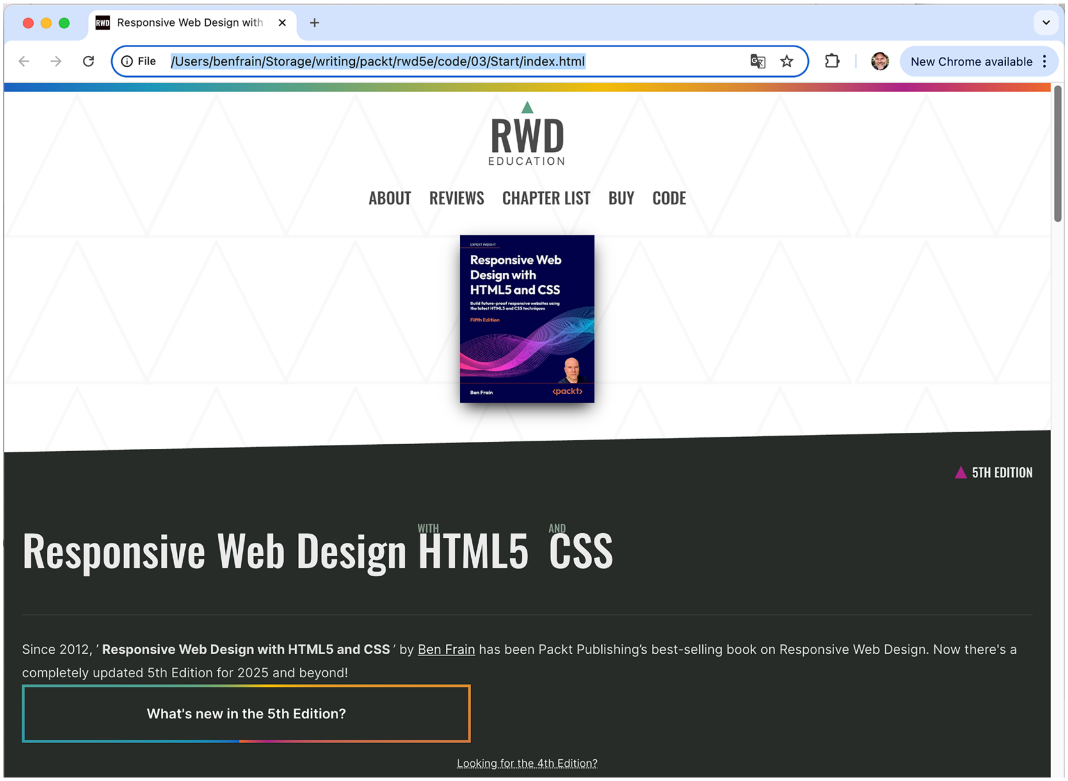
If you open the `index.html` file inside this chapter's `start` folder in a web browser, you will see that the design looks acceptable in slim viewports such as mobile phones:


09:58
RWD
EDUCATION
ABOUT
REVIEWS
CHAPTER LIST
BUY
CODE
Responsive Web Design with HTML5 and CSS
Fifth Edition
Ben Frain
‹packt›
5TH EDITION
Responsive Web Design
WITH
HTML5
AND
CSS
5e/code/03/Start/index.html
Looking for the 4th Edition?

Since 2012, ' **Responsive Web Design with HTML5 and CSS** ' by Ben Frain has been Packt Publishing's best-selling book on Responsive Web Design. Now there's a completely updated 5th Edition for 2025 and beyond!

Figure 3.1: The design looks fine in small viewports

However, it quickly starts to look a little stretched when you widen the browser window:


Responsive Web Design with
File /Users/benfrain/Storage/writing/packt/rwd5e/code/03/Start/index.html
New Chrome available
RWD
EDUCATION
ABOUT
REVIEWS
CHAPTER LIST
BUY
CODE
Responsive Web Design with HTML5 and CSS
5TH EDITION
Responsive Web Design WITH HTML5 AND CSS
What's new in the 5th Edition?
Looking for the 4th Edition?

Since 2012, ' **Responsive Web Design with HTML5 and CSS** ' by Ben Frain has been Packt Publishing's best-selling book on Responsive Web Design. Now there's a completely updated 5th Edition for 2025 and beyond!

Figure 3.2: We definitely need to change the styles for wider viewports

The initial styles are written using a fluid/proportional approach. This means widths are generally written as percentages rather than fixed pixel sizes, the upshot being that the content grows and shrinks to fill the browser window regardless of how large or small it gets.

However, this still leaves us with the problem that when you go wide enough, you need to do something a little more drastic, visually, than just making things wider still!

We can fix this problem with media queries.

Media queries allow us to target specific CSS styles depending on the capabilities of a device. For example, with just a few lines of CSS, we can change the way content is displayed based via things such as viewport width, screen aspect ratio, and orientation (landscape or portrait).

In this chapter, we will do the following:

- Understand the `meta` viewport tag to make media queries work on mobile devices
- Learn why media queries are essential for a responsive web design
- Understand the media query syntax
- Learn how to use media queries in links, in `@import` statements, and within CSS files
- Understand what device features we can test for
- Consider whether to put similar media queries in one block or wherever it suits
- Consider the latest capabilities added in Media Queries Level 5 that we can use today. Media features like `pointer`, `hover`, `prefers-color-scheme`, and `prefers-reduced-motion`, as well as better ways of dealing with viewport ranges
- Look at what future media queries may provide

Back in *Chapter 1*, *The Essentials of Responsive Web Design*, we inserted a `meta` tag into the head of our web page to make it work on mobile devices. At that point, I promised that *Chapter 3* would explain exactly what that tag is, what it does, and why we still have it. It's time to make good on that promise.

Remember that you can download the code for this chapter from the GitHub repository: `https://github.com/PacktPublishing/Responsive-Web-Design-with-HTML5-and-CSS-Fifth-Edition/tree/main/03`.

The meta viewport tag

When Apple released the iPhone in 2007, they introduced a proprietary `meta` tag called the `viewport meta` tag. Its purpose was to provide a way for web pages to communicate to mobile browsers how they would like the web browser to render the page.

Without this `meta` tag, iPhones would render web pages as a 980px wide window that the user would then have to zoom in or out of.

With this `meta` tag, it's possible to render a web page at its actual size and then adapt the layout to provide the kind of web page we now all expect to see when we browse the internet on our phones.

For the foreseeable future, any web page you want to be responsive and render well across small-screen devices will still need to make use of this `meta` tag as Android and a growing number of other platforms also support it.

As you are now aware, the viewport `meta` tag is added within the `head` tags of the HTML. It can be set to a specific width (which we could specify in pixels, for example) or as a scale, for example, 2.0 (twice the actual size). Here's an example of the viewport `meta` tag set to show the browser at twice (200 percent) the actual size:

```
<meta name="viewport" content="initial-scale=2.0,width=device-width" />
```

Let's break the above `meta` tag down and understand what's going on. The `name="viewport"` attribute tells the browser this tag is dealing with the viewport. The `content="initial-scale=2.0` section is then saying, "scale the content to twice the size" (where 0.5 would be half the size, 3.0 would be three times the size, and so on), while the `width=device-width` part tells the browser that the width of the page should be equal to `device-width`. The `meta` tag can also be used to control the amount a user can zoom in and out of the page. This example allows users to go as large as three times the device width and as small as half the device width:

```
<meta
    name="viewport"
    content="width=device-width, maximum-scale=3, minimum-scale=0.5"
/>
```

You could also disable users from zooming at all:

```
<meta name="viewport" content="initial-scale=1.0, user-scalable=no" />
```

Here, `user-scalable=no` is the relevant part.

However, with zooming being an important accessibility requirement for users with reduced visual acuity, it's rare that it would be appropriate to prevent it. In addition, many browsers disable the ability to prevent zooming for that very reason.

Right, we'll change the scale to 1.0, which means that the mobile browser will render the page at 100 percent of its viewport. Setting it to the device's width means that our page should render at 100 percent of the width of all supported mobile browsers. For the majority of responsive web design cases, this `meta` tag would be appropriate:

```
<meta name="viewport" content="width=device-width,initial-scale=1.0" />
```

That should be all you need to know about viewport `meta` tags from a responsive design perspective. In short, ensure you have added one or things are unlikely to look and behave as you might expect! To be certain everything is in order, ensure you test your code at some point on an actual device.

With the `meta` tag covered, we can now get to the real focus of the chapter: media queries themselves.

There are also newer useful meta tags like `<meta name="theme-color" content="#ff9900"/>` that provides a suggestion to the browser of how to color the surrounding UI. Typically this is the areas above you site on mobile browsers and can help give your content a more integrated appearance.

Why media queries are needed for a responsive web design

If you head over to the W3C specification of the CSS3 Media Queries module (`https://www.w3.org/TR/mediaqueries-3/`), you'll see that this is their official introduction to what media queries are all about:

> *A media query consists of a media type and zero or more expressions that check for the conditions of particular media features. Among the media features that can be used in media queries are 'width', 'height', and 'color'. By using media queries, presentations can be tailored to a specific range of output devices without changing the content itself.*

While a fluid layout, created with percentages rather than fixed widths, can carry a design a substantial distance (we cover fluid layouts in full in the next chapter), given the gamut of screen sizes we have to cover, there are times when we need to revise the layout more substantially. Media queries make this possible—think of them as basic conditional logic for CSS.

Basic conditional logic in CSS

True programming languages all have some facility in which one of two or more possible situations are catered for. This usually takes the form of conditional logic, also known as **control flow**, typified by an `if/else` statement.

If programming vernacular makes your eyes itch, fear not; it's a very simple concept. You probably dictate conditional logic every time you ask a friend to order for you when visiting a cafe, "If they've got triple chocolate muffins I'll have one of those; if not, I'll have a slice of carrot cake." It's a simple conditional statement with two possible (and equally desirable) results based upon two equally possible eventualities.

At the time of writing, CSS offers few possibilities when it comes to true conditional logic and programmatic features. Loops and iteration, for example, are still within the sole domain of CSS pre-processors (did I mention a fine book on the subject of the Sass pre-processor, called *Sass and Compass for Designers*?). However, media queries are one mechanism that does allow us to author basic conditional logic; styles are applied depending on whether certain media-based conditions are met.

Media query syntax

So, what does a CSS media query look like and, more importantly, how does it work?

Here is a complete but simple web page, with no content but some basic styles and media queries:

```html
<!DOCTYPE html>
<html class="no-js" lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Media Query Test</title>
        <meta name="viewport" content="width=device-width,
            initial-scale=1.0" />
        <style>
            body {
                background-color: grey;
            }
            @media screen and (min-width: 320px) {
                body {
                    background-color: green;
                }
            }
            @media screen and (min-width: 550px) {
                body {
                    background-color: yellow;
                }
            }
            @media screen and (min-width: 768px) {
                body {
                    background-color: orange;
                }
            }
            @media screen and (min-width: 960px) {
                body {
                    background-color: red;
                }
            }
        </style>
    </head>
    <body></body>
</html>
```

Copy and save that into a new file, or, if you have the downloaded code for the book, you can find it as `example_03-01`.

Now, open the file in a browser and resize the window. The background color of the page will vary depending on the current viewport size.

A default color is defined first, outside of a media query. Then, when any of the media queries are "true," the styles inside the media query overwrite the default.

The basic syntax of a media query is simple when you get used to it. You use the `@media` **at-rule** to communicate a media query and then write the media test in parentheses. The test must pass in order for the styles within the curly braces to be applied.

You can write media queries in links in HTML to load particular style sheets if the media query passes. You can write media queries on CSS `@import` at-rules to determine which style sheets should be imported. You can also write media queries directly into a CSS file to determine which rules should be applied on the basis of which media queries resolve to true.

Let's look at each in turn.

Media queries in link tags

Here's what a media query looks like on a link you'd find in the `<head>` section of markup:

```
<link
    rel="stylesheet"
    media="screen and (orientation: portrait)"
    href="portrait-screen.css"
/>
```

This media query is asking, "Are you a screen and is your orientation portrait?".

Media query on an @import at-rule

Here's the same rule on an `@import` statement:

```
@import url("portrait-screen.css") screen and (orientation: portrait);
```

You can see all the same components there: the file to be loaded and the test that has to be passed. Different syntax, same outcome.

Using the `@import` at-rule in CSS makes the browser request the relevant file from the network, and this can add to the number of HTTP requests. This can sometimes adversely affect site load speed.

Media queries in a CSS file

Finally, here's the same media query written inside a CSS file, or, within a `style` tag inside the HTML:

```
@media screen and (orientation: portrait) {
    /* styles here */
}
```

Inverting media query logic

It's possible to reverse the logic of any media query expression by adding `not` to the beginning of the media query. For example, the following code would negate the result in our prior example, loading the file for anything that wasn't a screen with a portrait orientation:

```
<link
    rel="stylesheet"
    media="not screen and (orientation: portrait)"
    href="portrait-screen.css"
/>
```

Although using the `not` keyword is occasionally useful, I find it is far simpler to just think about applying styles when you do want them. This way, you can stick to writing the most terse and simplest forms of media queries.

Combining media queries

It's also possible to string multiple expressions together. For example, let's extend one of our prior examples and also limit the file to devices that have a viewport that's larger than 800 pixels:

```
<link
    rel="stylesheet"
    media="screen and (orientation: portrait) and (min-width: 800px)"
    href="800wide-portrait-screen.css"
/>
```

A number of different media queries

We could also have a list of media queries. If any of the listed queries are true, the file will be loaded. If none are true, it won't. Here is an example:

```
<link
    rel="stylesheet"
    media="screen and (orientation: portrait) and (min-width: 800px),
        projection"
    href="800wide-portrait-screen.css"
/>
```

There are two points to note here. Firstly, a comma separates each media query, effectively acting like an `or` command. Secondly, you'll notice that after `projection`, there is no trailing `and` or feature/value combination in parentheses. That's because, in the absence of these values, the media query is applied to all media types. In our example, the styles will apply to all projectors.

You should be aware that you can use any CSS length unit to specify media queries. **Pixels** (`px`) are the most commonly used, but **ems** (`em`) and **rems** (`rem`) are equally valid and applicable.

Everyday media queries

At this point, it's probably prudent of me to tell you that in most situations, you don't actually need to specify `screen`. Here's the key point in the specification:

> *A shorthand syntax is offered for media queries that apply to all media types; the keyword 'all' can be left out (along with the trailing 'and'). I.e. if the media type is not explicitly given it is 'all'.*

Therefore, unless you want to target styles to particular media types, just leave the `screen and` part out. That's the way we will be writing media queries in the example files from this point on. Here's an example:

```
@media (min-width: 750px) {
    /* styles */
}
```

What can media queries test for?

When building responsive designs, the media queries that get used most usually relate to a device's viewport width (`width`). In my own experience, I have found little need, with the occasional exception of resolution and viewport height, to employ the other capabilities. However, just in case the need arises, here is a list of all capabilities that Media Queries Level 3 can test for. We will look at some other queries from later versions of the specification later in the chapter. For now, hopefully, some will pique your interest:

- `width`: The viewport width.
- `height`: The viewport height.
- `device-width*`: The rendering surface's width (for our purposes, this is typically the screen width of a device).
- `device-height*`: The rendering surface's height (for our purposes, this is typically the screen height of a device).
- `orientation`: This checks whether a device is portrait or landscape in orientation.
- `aspect-ratio`: The ratio of width to height based on the viewport width and height. A 16:9 widescreen display can be written as `aspect-ratio: 16/9`.
- `device-aspect-ratio*`: This is similar to `aspect-ratio` but is based on the width and height of the device's rendering surface, rather than the viewport.
- `color`: The number of bits per color component. For example, `min-color: 16` will check whether the device has a 16-bit color.
- `color-index`: The number of entries in the color lookup table (the table is how a device changes one set of colors to another) of the device. Values must be numbers and cannot be negative.
- `monochrome`: This tests how many bits per pixel are in a monochrome frame buffer. The value will be a number (integer), for example, `monochrome: 2`, and cannot be negative.
- `resolution`: This can be used to test screen or print resolution, for example, `min-resolution: 300dpi`. It can also accept measurements in dots per centimeter, for example, `min-resolution: 118dpcm`.
- `scan`: This can be either `progressive` or `interlace`, features largely particular to TVs. For example, a 720p HD TV (the "p" part of 720p indicates "progressive") could be targeted with `scan: progressive`, while a 1080i HD TV (the "i" part of 1080i indicates "interlaced") could be targeted with `scan: interlace`.
- `grid`: This indicates whether or not the device is based on a grid or bitmap.

All these features, with the exception of `scan` and `grid`, can be prefixed with `min` or `max` to create ranges. For example, consider the following code snippet:

```
@import url("tiny.css") screen and (min-width: 200px) and (max-width:
360px);
```

Here, a minimum (`min`) and maximum (`max`) have been applied to `width` to set a range. The `tiny.css` file will only be imported for screen devices with a minimum viewport width of 200 pixels and a maximum viewport width of 360 pixels.

*Features deprecated in CSS Media Queries Level 5

It's worth being aware that the draft specification for Media Queries Level 5, which in turn consumed the Media Queries Level 4 draft, deprecates the use of a few features (`https://www.w3.org/TR/mediaqueries-5/#mf-deprecated`), most notably `device-height`, `device-width`, and `device-aspect-ratio`. Support for those queries will remain in browsers but it's recommended you refrain from writing any new style sheets that use them.

Using media queries to alter a design

If you've read, and at least partially understood, what we have been through so far in this chapter, you should be ready to start using media queries in earnest.

I'd suggest opening the `index.html` file and the associated `styles.css` file from the `Start` folder of this chapter's code. Then try and add some media queries to alter some areas of the page at certain viewport widths. Let's make a couple of changes together, and then you can practice some of your own.

Let's look at the header section of what we have currently (viewport around 1200px wide):

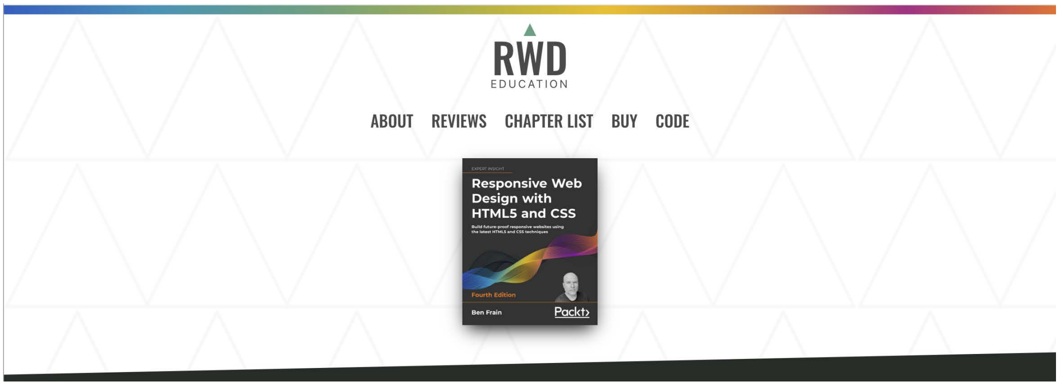


Figure 3.3: On wider viewports, what we have in the browser doesn't match the design

Here is the same section from the actual site:

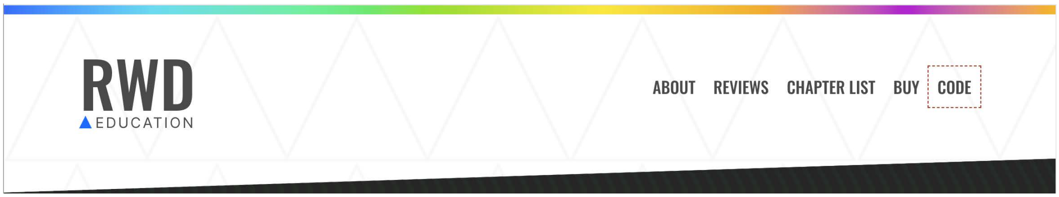

RWD
EDUCATION
ABOUT
REVIEWS
CHAPTER LIST
BUY
CODE

Figure 3.4: The design calls for a different layout at wider viewports

We need to keep the design as it is for small viewports but amend it for large ones. Let's start by adding a media query that makes the navigation links and the logo appear at opposite sides at 1200px and above:

```
@media (min-width: 1200px) {
    .rwd-MastHead {
        flex-direction: row;
        justify-content: space-between;
        max-width: 1000px;
        margin: 0 auto;
    }
}
```

If any of those styles don't make sense now, don't worry. We will be covering Flexbox layout in *Chapter 4*, *Fluid Layout and Flexbox*. What is important to understand is the media query itself. What we are saying is "apply this rule, but only at a minimum width of 1200px."

Here is the effect of that rule in the browser:

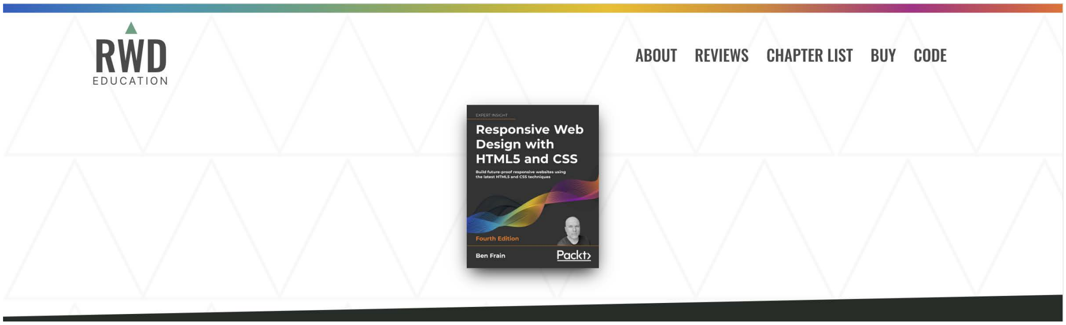
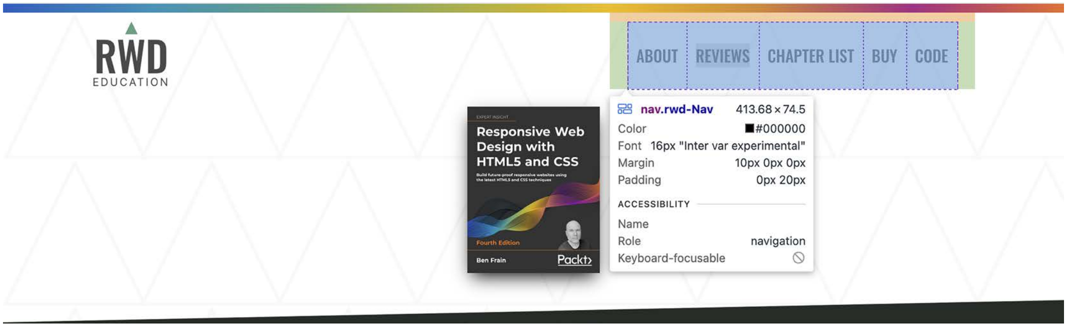

RWD
EDUCATION
ABOUT
REVIEWS
CHAPTER LIST
BUY
CODE
Responsive Web Design with HTML5 and CSS
Fourth Edition
Ben Frain
Packt

Figure 3.5: Our first media query puts the logo on the left and the menu on the right

Now, the keen-eyed among you may realize that despite this new rule, which spreads out logo and navigation links, we have applied some styles that we no longer need at this size. The following screenshot shows the no-longer-needed margin above and inline padding within the main navigation links:



nav.rwd-Nav 413.68 × 74.5
Color #000000
Font 16px "Inter var experimental"
Margin 10px 0px 0px
Padding 0px 20px
ACCESSIBILITY
Name
Role navigation
Keyboard-focusable

Figure 3.6: There are styles being applied that we want to reset for this viewport

Let's write another media query to fix this. We'll make this change at a slightly smaller screen width, as that extra space is unneeded even at 1,000px wide:

```css
@media (min-width: 1000px) {
    .rwd-Nav {
        margin: 0;
        padding: 0;
    }
}
```

In this instance, we are saying "at 1,000px and above, make the margin and padding zero for the `.rwd-Nav` element." Notice that each time, in our media query, we only amend the specific properties we need to change. By writing our "base" styles as we have, outside of any media query, our actual media queries only need to encapsulate the differences needed.

It might seem a little reductive, but this process is essentially all there is to working with media queries: write the basic styles without media queries and then work wider and wider, adding media queries and changes wherever needed to affect the design as required.

If you have the time, I implore you to try writing some media queries for yourself. Whether it's the test page here or something of your own that you want to work with, pick something you want to change, pick a viewport width, and add a media query to make the change happen. Once you have written a few, making the changes you want to see, you will realize how straightforward but powerful they are.

Testing responsive designs on emulators and simulators

Although there is no substitute for testing your development work on real devices, there are emulators for Android and a simulator for iOS (for the pedantic, a simulator merely simulates the relevant device, whereas an emulator actually attempts to interpret the original device's code). The Android emulator for Windows, Linux, and Mac is available for free by installing Android Studio from `https://developer.android.com/studio`. The iOS simulator is only available to macOS users and comes as part of the Xcode package (free from the Mac App Store). Browsers themselves also include ever-improving tools for emulating mobile devices in their development tools.

Advanced media query considerations

The following section deals with concerns for when you are highly proficient in writing media queries. Think of these topics as micro-optimizations. If you are just starting with media queries, you certainly shouldn't be worrying about any of these topics yet. Jump to the *Media Queries Level 5* section instead!

OK, media query uber-geekery, here we go…

Do media queries affect which files get downloaded or how the page renders?

Ordinarily, for a browser, CSS is considered to be a **render-blocking** asset; the browser needs to fetch and parse a linked CSS file before the rendering of the page is complete. This stands to reason, as the browser needs to know what styles to apply for laying out and painting the page. However, modern browsers are smart enough to discern which style sheets (linked with media queries in the head) need to be analyzed immediately and which can be deferred until after the initial page rendering. The upshot of this is that for these browsers, CSS files that are linked with media queries that don't apply to the current environmental situation can be *deferred* until after the initial page load, providing some performance advantage. There's more on this topic over on Google's developer pages: `https://web.dev/articles/critical-rendering-path/render-blocking-css`.

To re-iterate, all the linked files will still be downloaded; they just may not necessarily require the browser to hold the rendering of the page.

Therefore, a modern browser loading a responsive web page with four different style sheets linked with different media queries (to apply different styles for different viewport ranges) will download all four CSS files but will probably only parse the applicable one initially before rendering the page. Take a look at `example_03-02` in the chapter's example code, which you may find more convenient to view at `https://benfrain.com/playground/mq-downloads/`. If you view that page with a slim viewport and you are proficient enough with your browser's developer tools, you can look in the network area and check that all files are downloaded, regardless of whether the media queries are actually being applied.

Should you split media queries into their own files?

Let's suppose you have a style sheet with many different media queries. Suppose there are several for min-widths of 800px and several more for widths above 1,200px. Should you separate them into their own style sheets? In short, I would say no.

Apart from preference and/or compartmentalization of code, there is rarely a great tangible advantage in separating different media query styles into separate files. After all, using separate files increases the number of HTTP requests needed to render a page, which in turn can make the pages slower in certain other situations. Nothing is ever easy on the web! It is, therefore, really a question of evaluating the entire performance of your site and testing each scenario on different devices.

As a default approach, unless the project has considerable time available for performance optimizations, this is one of the last places I would look to make performance gains.

More practically, only once I am certain all images are compressed, all scripts are squished as small as possible with concatenation and minification, all assets are being served gzipped, all static content is being cached via CDNs, and all surplus CSS rules have been removed would I start looking to split up media queries into separate files for potential performance gains.

In all but exceptional circumstances, I recommend adding media queries within an existing style sheet alongside the "normal" rules. If you are happy to do so, that leads to one further consideration: should media queries be declared underneath the associated selector or split off into a separate block of code at the end for all similar media queries? I'm glad you asked.

Consolidate media queries or scatter them where it suits?

I'm a fan of writing media queries underneath the original "normal" rule. For example, let's say I want to change the width of a couple of different elements, at different places in the style sheet, depending on the viewport width. I would typically do this:

```
.thing {
    width: 50%;
}
@media (min-width: 30rem) {
    .thing {
        width: 75%;
    }
}
/* More styles for different elements */
.thing2 {
    width: 65%;
}
@media (min-width: 30rem) {
    .thing2 {
        width: 75%;
    }
}
```

Can you see in this example that we have two separate media queries that are testing for the same thing: `@media (min-width: 30rem)`? Surely duplicating media at-rules like this is overly verbose and wasteful? Surely I should be advocating grouping all the like media queries into a single block like this:

```
.thing {
    width: 50%;
}
.thing2 {
    width: 65%;
}
/* Styles affecting other elements */
@media (min-width: 30rem) {
    .thing {
        width: 75%;
    }
    .thing2 {
        width: 75%;
    }
}
/* A few more styles go after */
```

That is certainly one way to do it. However, from a maintenance point of view, I find this more difficult. There is no "right" way to do this, but my preference is to define a rule for an individual selector once and have any variations of that rule (such as changes within media queries) defined immediately after. That way, I don't have to search for separate blocks of code to find the declaration that is relevant to a particular selector.

Nesting media and container queries

With CSS preprocessors, or using native CSS nesting, which we cover in *Part 2* of this book, associating all similar rules can be even more convenient by nesting media queries directly within the initial ruleset. This is my preference as it means the key selector of the rule is the "source of truth," and all eventualities for that selector are contained within a set of braces. That sounds rather grandiose! Hopefully, the following code demonstrates the same rules as before but nested:

```
.thing {
    width: 50%;
    @media (min-width: 30rem) {
        width: 75%;
    }
}

.thing2 {
    width: 65%;
    @media (min-width: 30rem) {
        width: 75%;
    }
}
```

What about file size?

If you aren't nesting, it would seem fair to argue against having duplicated `@media` rules on the grounds of verbosity. Surely file size alone should be enough reason to not write media queries in this manner? After all, no one wants a big, bloated CSS file to serve their users. However, the simple fact is that gzip compression, which should be compressing all possible assets on your server, reduces the difference to a completely inconsequential amount.

I've done various tests on this in the past, so if it's something you would like to read more about, head over to `http://benfrain.com/inline-or-combined-media-queries-in-sass-fight/`. The bottom line is: I don't believe you should concern yourself with file size if you would rather write media queries directly after the "standard" styles.

As a final note on this, if you want to author your media queries directly after the original rule but have all similar media query definitions merged into one, there are a number of build tools that facilitate this.

That should give you everything you need to start wielding media queries like a pro. However, before we move on, there are a number of media query features in Media Queries Level 5 we can actually start using today. Let's take a sneak peek!

Media Queries Level 5

Specifications at the W3C go through a ratification process, from **Working Draft** (**WD**) to **Candidate Recommendation** (**CR**) to **Proposed Recommendation** (**PR**), before finally arriving, many years later, at W3C **Recommendation** (**REC**). So, modules at a greater maturity level than others are generally safer to use. For example, CSS Spatial Navigation Level 1 is in progress as I write this (`http://www.w3.org/TR/css-nav-1/`), at Working Draft status with no support in browsers. Meanwhile, the topic of this chapter until this point has been Media Queries Level 3, which is implemented in every modern browser.

> If you have a spare day, you can knock yourself out reading all about the official explanation of the standards ratification process at `http://www.w3.org/2005/10/Process-20051014/tr`.

At the time of writing, while CSS Media Queries Level 5 has a draft specification (`https://www.w3.org/TR/mediaqueries-5`), not all the features it documents have browser implementations. So, in this section, we will concentrate on Media Queries Level 5 features that we can make use of: features already implemented in browsers.

Interaction media features

Interaction media queries are concerned with pointing devices and hover capability. Let's see what each of these can do for us.

The pointer media feature

Here is the W3C introduction to the `pointer` media feature:

> *The pointer media feature is used to query the presence and accuracy of a pointing device such as a mouse. If multiple pointing devices are present, the pointer media feature must reflect the characteristics of the "primary" pointing device, as determined by the user agent.*

There are three possible states for the `pointer` features: `none`, `coarse`, and `fine`.

A `coarse` pointer device might be a finger on a touchscreen device. However, it could equally be a cursor from a games console that doesn't have the fine-grained control of something like a mouse.

```
@media (pointer: coarse) {
    /* styles for when coarse pointer is present */
}
```

A `fine` pointer device might be a mouse but could also be a stylus pen or any future fine-grained pointer mechanism.

```
@media (pointer: fine) {
    /* styles for when fine pointer is present */
}
```

Browsers report whether the value of `pointer` is `fine`, `coarse`, or `none`, based on the "primary" pointing device. Therefore, consider that just because a device has the capability of a fine pointer, it doesn't mean that will be the primary pointing device. Think of tablets where the primary pointer is a finger (`coarse`), but that has an attached stylus (a `fine` pointing device).

> The safest bet is always to assume users are using touch-based input and size user interface elements accordingly. That way, even if they are using a mouse, they will have no difficulty using the interface with ease. If, however, you assume mouse input and don't provide affordance for coarse pointers, it might make for a difficult user experience.

Read the draft of this feature here: `https://www.w3.org/TR/mediaqueries-5/#pointer`.

The hover media feature

As you might imagine, the `hover` media feature tests the device's ability to hover over elements on the screen. If the user has multiple inputs at their disposal (touch and mouse, for example), characteristics of the primary input are used. Here are the possible values and example code:

For users that have no ability to hover, we can target styles at them with a value of none:

```
@media (hover: none) {
    /* styles for when the user cannot hover */
}
```

Or, as before, we might choose to make the non-hover scenario the default and then only add hover styles for devices that take advantage of them:

```
@media (hover) {
    /* styles for when user can hover */
}
```

Be aware that there are also `any-pointer` or `any-hover` media features. They are like the preceding `hover` and `pointer` but test the capabilities of any of the possible input devices.

That way, if you want to apply styles if any input device is capable of hover, regardless of whether that input device is the primary one, you can:

```
@media (any-hover: hover) {
    /* styles if any input device is capable of hover */
}
```

If you want to style an element a certain way based on whether any attached pointer device was coarse, you could use `any-pointer` like this:

```
@media (any-pointer: coarse) {
    /* styles to be applied if any attached pointer is coarse */
}
```

User preference media features

There are a number of features in the Media Queries Level 5 draft that cater to the preferences and needs of users. Some of them can be seen as largely superficial, while others have more concrete applications, such as enhancing accessibility. Let's look at some next.

The prefers-color-scheme media feature

In more recent years, popular operating systems for both desktop and mobile computers have given users the option of a `dark mode`. To supplement this, operating systems expose this user preference to the browser by way of the `prefers-color-scheme` media feature.

At present, there are three possible preferences: `light`, `dark`, and `no-preference`. To exemplify this feature's use, we might amend the default colors for a page like this:

```css
body {
    background-color: #e4e4e4;
    color: #545454;
}
@media (prefers-color-scheme: dark) {
    body {
        background-color: #333;
        color: #ddd;
    }
}
```

In the same vein as I'm recommending you to write your default "mobile" styles in the root of your style sheet, I recommend writing the default colors in the root too and adding one of these queries to cater for an alternative interface if needed or desired.

You can read the draft specification for the `prefers-color-scheme` media feature here: `https://www.w3.org/TR/mediaqueries-5/#prefers-color-scheme`.

The prefers-reduced-motion feature

If a user has set their device to minimize the amount of motion it provides in its interface and system interactions, this information is exposed to CSS via the `prefers-reduced-motion` media query.

There are only two settings: `no-preference` or `reduce`.

If you find yourself writing more than the occasional animation or transition, it's well worth wrapping such styles up in a `no-preference` variant like this:

```css
@media (prefers-reduced-motion: no-preference) {
    /* animation code here */
}
```

If you want to provide a different, more subtle animation to your users, you could write your animation code and then provide another that overwrites the first for users who would like the more subtle variant:

```css
.ele {
    animation: bezerk 1s;
}
```

```
@media (prefers-reduced-motion) {
    .ele {
        animation: subtle 1s;
    }
}
```

Note in the example code that when not explicitly stating a preference, merely using the `prefers-reduced-motion` query will evaluate to true.

However you prefer to write them, this media query provides a great tool to cater to users sensitive to animations and movement, whether for medical reasons such as vestibular motion disorders or purely from a personal preference point of view.

Future possibilities

A glance down the Level 5 draft specification reveals a multitude of additional features that we haven't discussed here. There is the ability to detect whether scripting is supported (think JavaScript) and the ability to create `custom` media queries; don't get too carried away, it's only really a nicer way to write them.

Range syntax

Since 2023, we have also been able to specify ranges in a terser syntax. For example, instead of this:

```
@media (min-width: 30rems) and (max-width: 60rems) {
    /* Styles */
}
```

You can write this:

```
@media (30rems <= width <= 60rems) {
    /* Styles */
}
```

I don't know if it's a lack of exposure, but I find that new syntax quite difficult to parse. Simpler queries like this one are arguably fairly straightforward to understand:

```
@media (width >= 800px) {
    /* Styles */
}
```

I don't personally feel the minor saving of characters is worth it over the more self-explanatory version:

```
@media (min-width: 800px) {
    /* Styles */
}
```

But I suppose it is good to have the option to choose your preference.

Container queries

Media queries operate their queries on the capabilities of the device. Practically, when working with responsive designs, it's most commonly the width of the viewport we are interested in.

However, there are occasions where the width of the viewport is not the ideal metric for making a style change.

Suppose we are creating a visual component that might be used in a wider container, such as a `main` content area, but equally might be used in a thinner, sidebar `aside` area. These two contextual conditions for our component can exist at the same time. The viewport width would be the same, but the container our component finds itself in would be different. This is the problem that container queries solve. They allow us to make styles based on the container size, rather than the viewport size.

Before we look at container queries in more depth, let's consider what we did before them, not just to reminisce, but because container queries are a relatively new addition to the web platform, and knowing how to work without them may prove useful.

We tended to work around these scenarios without container queries by using descendant selectors to give some context and make style changes based on the document structure:

```
.component {
    font-size: 16px;
}

.sidebar .component {
    font-size: 14px;
}
```

Then, with this override established for when the component is in the sidebar, we can combine these selectors with media queries as needed. Maybe we need one set of styles when in the `.sidebar` and the screen is a certain size, and then another set of styles when not in the `.sidebar` but we are still at that viewport size:

```
@media (min-width: 40em) {
    .component {
        /* styles */
    }
    .sidebar .component {
        /* styles */
    }
}
```

That works, but in this scenario, we need to be very clear about the environments our piece of user interface may find itself in. It's also quite a broad stroke. What happens when the structure or physical shape of the rendered document is changed, as can happen in responsive designs?

As discussed at the beginning of this section, what we are actually interested in is the width of the container our component finds itself in and container queries provide a cleaner and more robust mechanism to do that.

In 2022, container queries started to make their way into browsers.

Using container queries

Media queries are always available, due to there always being some media you are viewing the website on. That is not the case for container queries. For container queries, it is necessary to tell the browser which elements we would like to be query-able containers. Effectively, this is like saying to the browser, "Hey, see this element here? I might want to know how wide that is." Therefore, we need to learn how to tell the browser which elements are containers so that we can query them.

In addition, in order for a container to be query-able, we also need to tell the browser how the container should be "contained."

As we have established, typically, with a responsive design, width is the axis we are interested in querying. For container queries, this is known as "inline-size," which technically looks at the length of the container in the inline axis. We can also opt to set up a container to be queried in both inline or block. In a left-to-right, top-to-bottom Western language such as English, all you need to know is that inline is horizontal, and block is vertical.

Let's consider a contrived scenario. You can view this yourself by looking at `example_03-03`.

We will make ourselves a box, containing some text. The text in the box will get larger if the box's container is larger and smaller when the box's container is smaller.

To further illustrate this, the box itself will be moved about the page in one of three positions, depending upon the width of the viewport. However, in two of those positions, the box, and therefore the text within, will actually get smaller, despite the viewport being far wider in those scenarios. We are also changing the background color of the element for the different states of the container query. Let's take a look at the CSS. Concentrate on the `containing-element` and `contained-content` rules here (don't worry about the grid-related declarations; we will cover grids in detail in *Chapter 5*):

```
main {
    display: grid;
    grid-template-columns: auto;
    @media (min-width: 800px) {
        grid-template-columns: 1fr 450px 1fr;
    }
    @media (min-width: 1000px) {
        grid-template-columns: 1fr 550px 300px;
    }
}

.containing-element {
    container: limerick / inline-size;
    grid-column: 1;
    @media (min-width: 800px) {
        grid-column: 2;
    }
    @media (min-width: 1000px) {
        grid-column: 3;
    }
}

.contained-content {
    padding: 10px;
    margin: 0;
```

```
        font-size: 18px;
        background-color: #ff000099;
        @container (min-width: 400px) {
            font-size: 1.5em;
            background-color: #00ff0099;
        }
        @container (min-width: 500px) {
            font-size: 2em;
            background-color: #0000ff99;
        }
    }
```

The markup looks like this:

```
<main>
            <aside class="containing-element">
                <p class="contained-content">
                    There once was a man from Brazil, Who ate a
                    dynamite pill...
                </p>
            </aside>
        </main>
```

Here, `containing-element` gets moved around the page and subsequently changes size. The text inside `contained-content` is using a container query that allows it to change font size and background based on the size of `containing-element`.

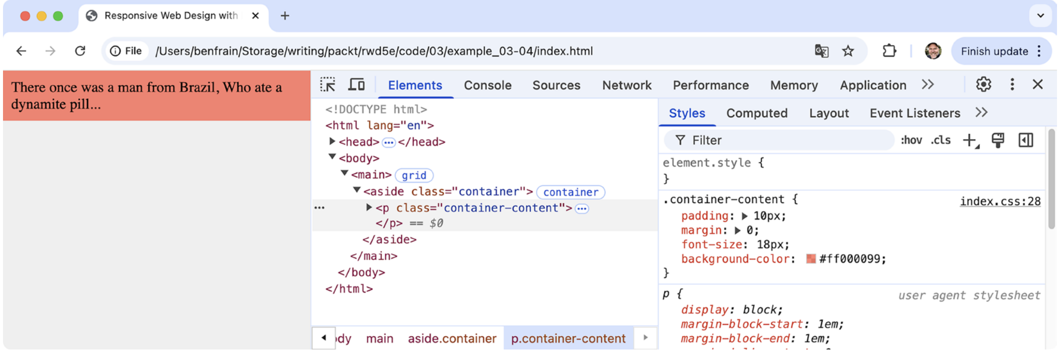
Open that in a browser and resize the browser window, and you will see the container moving position inside the wider grid, which subsequently gives it a different size.

The `container` property we have used in the example code is actually a shorthand for setting two other properties: `container-name` and `container-type`. You can set them separately using those properties if you prefer. But here, we are setting a name of `limerick` for the container and a type of `inline-size` to allow us to do something on the basis of its width. Remember earlier, when we said you can also query the height? If you wanted to do that, you would change the line to container: `limerick / size`.

With those properties set on the container and the elements nested, such as our `.contained-content` class, can now query the container.

In our example, we effectively have three states for the content of our `contained-content`: one with no container query active, one container query that is true if the container is at least 400px wide, and another container query that is true if the container is at least 500px wide. In each, there is a different `font-size` and `background-color`.

Open `example_03-03` and try resizing the browser. Here is the viewport at 375px wide. Notice that the text is set to 18px and the background color is red.


Responsive Web Design with
File /Users/benfrain/Storage/writing/packt/rwd5e/code/03/example_03-04/index.html
Finish update
Elements
Console
Sources
Network
Performance
Memory
Application
Styles
Computed
Layout
Event Listeners
Filter
:hov .cls
›dy main aside.container p.container-content

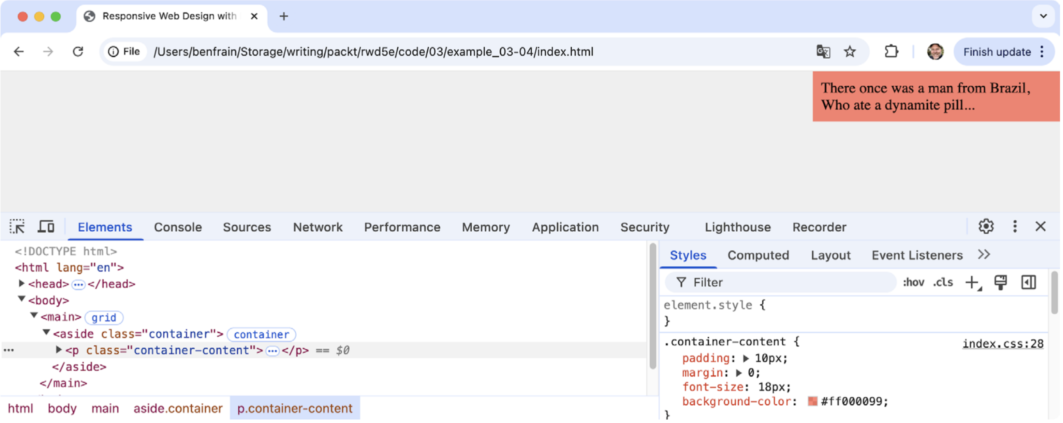
There once was a man from Brazil, Who ate a dynamite pill...

```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
 ▼<body>
   ▼<main> grid
     ▼<aside class="container"> container
       ▶<p class="container-content">…
        </p> == $0
      </aside>
    </main>
  </body>
</html>
```

```
element.style {
}
.container-content {                       index.css:28
  padding: ▶ 10px;
  margin: ▶ 0;
  font-size: 18px;
  background-color: #ff000099;
}
p {                               user agent stylesheet
  display: block;
  margin-block-start: 1em;
  margin-block-end: 1em;
```

Figure 3.7 – The viewport and therefore the container here is just 375px

Here is a far wider viewport at 1,015px, and notice that the text remains the same because in this scenario, the container is still less than 400px.


Responsive Web Design with
File /Users/benfrain/Storage/writing/packt/rwd5e/code/03/example_03-04/index.html
Finish update
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
Recorder
Styles
Computed
Layout
Event Listeners
Filter
:hov .cls
html body main aside.container p.container-content

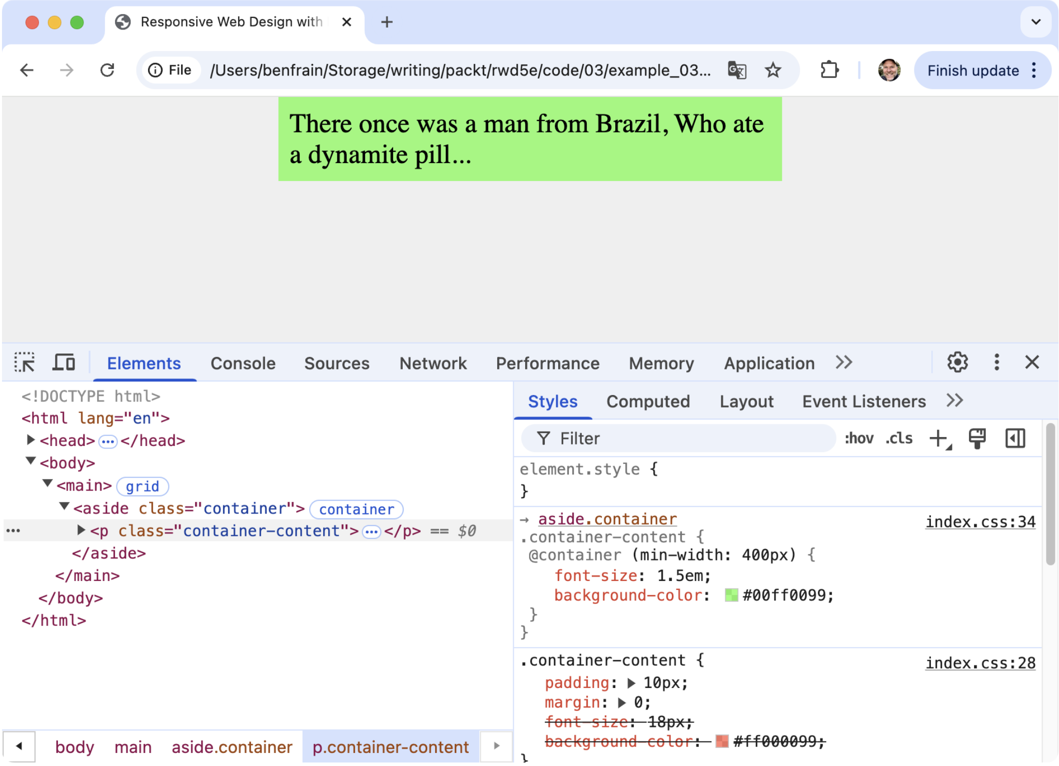
There once was a man from Brazil, Who ate a dynamite pill...

```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
 ▼<body>
   ▼<main> grid
     ▼<aside class="container"> container
       ▶<p class="container-content">…</p> == $0
      </aside>
    </main>
```

```
element.style {
}
.container-content {                       index.css:28
  padding: ▶ 10px;
  margin: ▶ 0;
  font-size: 18px;
  background-color: #ff000099;
}
```

Figure 3.8 – Wide viewport but the container is still under 400px wide

However, if we set the viewport to say 950px, the container here is actually wider, so despite the viewport being smaller, the container query for a *min-width* of 400px is true, the text is larger, and the background is green.


Responsive Web Design with
File
/Users/benfrain/Storage/writing/packt/rwd5e/code/03/example_03...
Finish update
Elements
Console
Sources
Network
Performance
Memory
Application
Styles
Computed
Layout
Event Listeners
Filter
:hov
.cls
body
main
aside.container
p.container-content

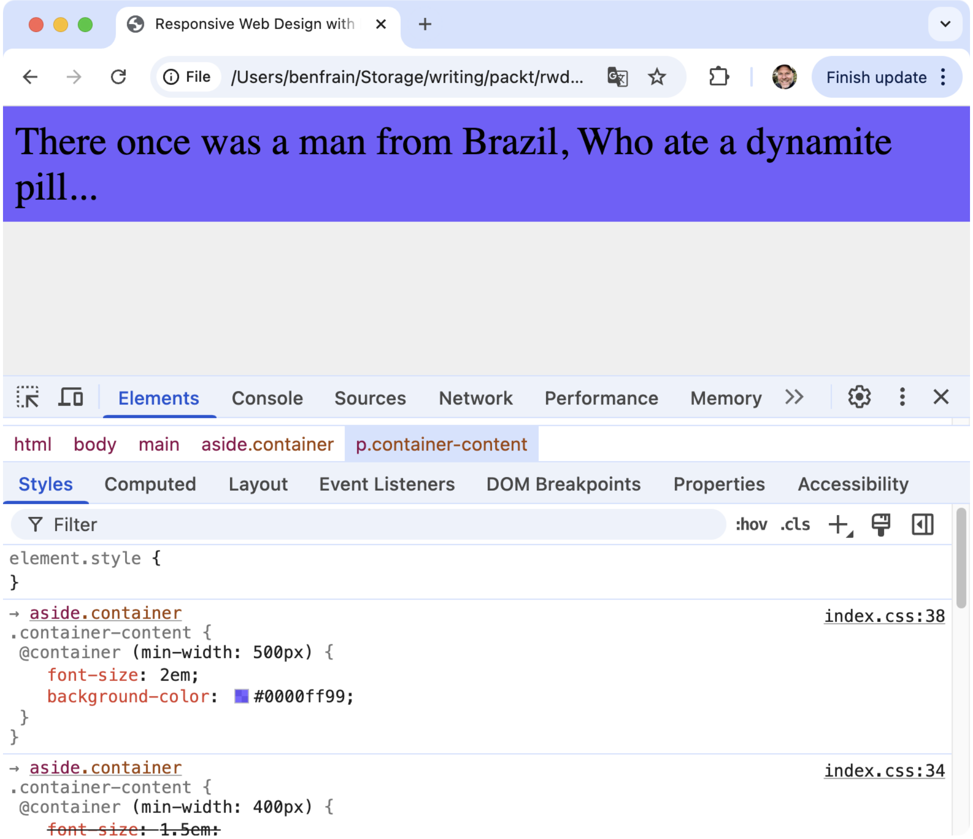
There once was a man from Brazil, Who ate a dynamite pill...

```
<!DOCTYPE html>
<html lang="en">
 ▶<head> ··· </head>
 ▼<body>
   ▼<main> grid
     ▼<aside class="container"> container
···    ▶<p class="container-content"> ··· </p> == $0
      </aside>
    </main>
  </body>
</html>
```

```
element.style {
}
→ aside.container                                index.css:34
.container-content {
 @container (min-width: 400px) {
    font-size: 1.5em;
    background-color: 🟩 #00ff0099;
 }
}
.container-content {                             index.css:28
   padding: ▶ 10px;
   margin: ▶ 0;
   ~~font-size: 18px;~~
   ~~background-color: 🟥 #ff000099;~~
}
```

Figure 3.9 – Small viewport but large container

And if we make the viewport smaller still, to around 780px, the container is changed in the layout so it is larger still and we get larger text and a blue background.


Responsive Web Design with
File
/Users/benfrain/Storage/writing/packt/rwd...
Finish update
Elements
Console
Sources
Network
Performance
Memory
html
body
main
aside.container
p.container-content
Styles
Computed
Layout
Event Listeners
DOM Breakpoints
Properties
Accessibility
Filter
:hov
.cls
index.css:38
index.css:34

There once was a man from Brazil, Who ate a dynamite pill...

```
element.style {
}
→ aside.container
.container-content {
 @container (min-width: 500px) {
    font-size: 2em;
    background-color: #0000ff99;
 }
}
→ aside.container
.container-content {
 @container (min-width: 400px) {
    ~~font-size: 1.5em;~~
```

Figure 3.10 – Medium-sized viewport has the widest container

Why name the container?

If there were no other ancestor containers, then we wouldn't need to bother with a named container. We could just write a container query like this:

```
@container (inline-size > 500px) { /* styles */ }
```

Note that I have used the alternative syntax of `inline-size` rather than `min-width`, just to remind you that you can!

However, if there is a possibility of more than one ancestral container, this approach may become problematic.

`container-name` allows us to write container queries that are specific to a specific container. So, assuming that an imaginary component has two ancestral containers, one called `main` and the other called `sidebar`, we can have some styles applied when the `main` container is over 500px wide like this:

```
@container main (inline-size > 500px) {
    .thing {
        /* styles */
    }
}
```

Those styles will only be applied when the `main` container is more than 500px wide and, crucially, *not* if the `sidebar` container is more than 500px wide.

By default, container queries resolve to their nearest ancestor container that accepts querying on the specified axis, or the smallest viewport size if none are available.

As named containers provide the ability to query more than one container, we can effectively say, "If `main` is 400px wide, apply one set of styles, and if `sidebar` is over 20rem high, apply these other styles." This provides a lot of flexibility.

Container relative lengths

We can change the size of the child elements of containers using container relative lengths.

Here are the units:

- `cqi`, 1% of the inline-size
- `cqw`, 1% of the width
- `cqb`, 1% of the block-size
- `cqh`, 1% of the height
- `cqmin`, 1% of the smallest of `cqi` and `cqb`
- `cqmax`, 1% of the largest of `cqi` and `cqb`

These resolve to the nearest ancestor container that accepts queries on that axis. This is an important concept to fully grasp. Let's assume you specified `40cqh`. If the nearest ancestor container is set to `inline-size`, then it won't be 40% of the height of that container. It will be the nearest ancestor container set for `size`.

However, knowing that we can size things based on the size of the container, maybe we could rewrite our prior example to use container relative lengths for the font sizing?

As an exercise, try removing the three existing font-size rules and just have a single one that sets font-size based on the container's inline-size.

Possibilities

If you had a go at changing the prior example to use container relative lengths for font-size, you will already know how easy it is to scale text to the size of the container. We will look at CSS functions such as `calc`, and `clamp` in later chapters but incredibly responsive typography is possible by combining some base font sizing, plus some container-related length value. That would be as simple as this:

```
.heading {
    font-size: calc(14px + 2cqi);
}
```

Remember, we will be able to make all these styling forks on containers independently or in conjunction with styling forks going on with the viewport.

Container style queries

Besides container *size* queries, we also have container *style* queries. The promise of style queries is that they will enable you to fork your code on, say, whether or not `background-color` is a certain value, or whether the text is a certain size. That would look like this:

```
@container main style(background-color: #ff9900) or style(font-size: 10px)
{
    .inner-text {
        color: #222;
    }
}
```

In a similar manner to the way that media queries, container queries, and feature queries work, style queries allow you to chain and negate any number of style queries on a container. The `style()` function evaluates to true or false, and the related styles therefore applied or not.

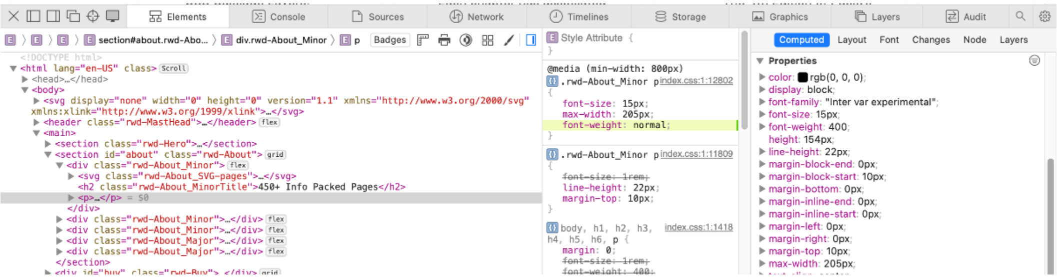
You can read the specification for container style queries here: `https://www.w3.org/TR/css-contain-3/#container-style-query`.

There are some important things to understand when using style queries.

Style queries work on the computed value of a property

Style queries work on the computed style of a property. This means that matching for bold on `font-weight`, as an example, would resolve true if you had set font-weight to 700 because bold and 700 are equivalent when computed. The same is true for colors and different syntaxes, such as `rgb`, `hsl`, and `hex`, and any other value that has different syntaxes that ultimately compute to the same value.

The developer tools of every browser include a computed tab for the styles panel. This shows you what the values that have been applied via styles have actually been computed as by the browser. You can test this for yourself by setting some text to have a font-weight of normal in your CSS and then loading the page up in a browser, selecting the element with dev tools, and checking the computed tab of the styles panel. You will probably see that font-weight was computed to a value of 400.



```
<!DOCTYPE html>
▼ <html lang="en-US" class> Scroll
  ▶ <head>…</head>
  ▼ <body>
    ▶ <svg display="none" width="0" height="0" version="1.1" xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">…</svg>
    ▶ <header class="rwd-MastHead">…</header> flex
    ▼ <main>
      ▶ <section class="rwd-Hero">…</section>
      ▼ <section id="about" class="rwd-About"> grid
        ▼ <div class="rwd-About_Minor"> flex
          ▶ <svg class="rwd-About_SVG-pages">…</svg>
            <h2 class="rwd-About_MinorTitle">450+ Info Packed Pages</h2>
          ▶ <p>…</p> = $0
          </div>
        ▶ <div class="rwd-About_Minor">…</div> flex
        ▶ <div class="rwd-About_Minor">…</div> flex
        ▶ <div class="rwd-About_Major">…</div> flex
        ▶ <div class="rwd-About_Major">…</div> flex
        </section>
```

```
Style Attribute {
}

@media (min-width: 800px)
.rwd-About_Minor p                index.css:1:12802
{
  font-size: 15px;
  max-width: 205px;
  font-weight: normal;
}

.rwd-About_Minor p                index.css:1:11809
{
  font-size: 1rem;
  line-height: 22px;
  margin-top: 10px;
}

body, h1, h2, h3,                 index.css:1:1418
h4, h5, h6, p {
  margin: 0;
  font-size: 1rem;
  font-weight: 400;
```

```
▶ color: ■ rgb(0, 0, 0);
▶ display: block;
▶ font-family: "Inter var experimental";
▶ font-size: 15px;
▶ font-weight: 400;
  height: 154px;
▶ line-height: 22px;
▶ margin-block-end: 0px;
▶ margin-block-start: 10px;
▶ margin-bottom: 0px;
▶ margin-inline-end: 0px;
▶ margin-inline-start: 0px;
▶ margin-left: 0px;
▶ margin-right: 0px;
▶ margin-top: 10px;
▶ max-width: 205px;
```

Figure 3.11 You can check the value anything computes to in the styles panel of dev tools

Nesting container style queries

You can nest container style queries inside other container style queries. But I'm going to say here that if you need to do this or choose to do this, you probably need to stop and think harder about what you are doing in the first place as this might well be symptomatic of complex logic that could be simplified.

That would look something like this (you can view this in `example-03-04`):

```
@container style(background-color: goldenrod) {
    @container style(color: #333) {
        .nested {
            font-size: 1.5em;
        }
    }
}
```

The element with a class of nested would have a font-size of 1.5em if it was nested inside a container that had color set to #333, which in turn was inside a container that had its `background-color` set to goldenrod.

For the time being, the biggest caveat of using container style queries is that they are only supported with CSS custom properties.

Currently only work with CSS custom properties

I deliberately started this section on container queries with the words 'The promise…' because at the time of writing, style queries are limited to only being usable with CSS custom properties.

We haven't covered custom properties yet, but we will cover them fully in *Chapter 13*, *Custom Properties and CSS Functions*. Don't worry, though; all you need to know for now is that a custom property is a means of assigning a value that you want to re-use to a named token. That isn't all they are for, but in relation to container style queries, that basic knowledge will suffice.

Consider this CSS code:

```
:root {
    --bg: black;
    font-size: 1.5rem;
}

.outer {
    background-color: var(--bg);
    container-name: outer;
}

.inner {
    @container outer style(--bg) {
            color: red;
        }
}
```

`:root` is our document, and then we have just two elements for some simple markup:

```
<div class="outer">
    <div class="inner">
        What color will I be?
    </div>
</div>
```

The double dash notation for `--bg` in the root is us setting a custom property to the value black.

On the outer element, which has a `container-name` of `outer`, we use that custom property of `--bg` as the value of `background-color`.

Our inner element has a container style query that checks for the presence of that custom property on a container. Here, we are not checking what that custom property is set to, although we could; we are merely querying that it is set.

Before you open this example in `example_03-06`, what color do you think the text will be, and what color do you think the background will be?

What about if, on the style query, you did not use the name of a container? We have it set to look at outer. What would you expect to happen if you removed that?

Now take a look at the example in a browser, change those values, and check your assumptions.

Container queries open yet more possibilities to us with CSS. I'll be honest and say at this point that I have found little practical use for them, but it may purely be that I am used to solving problems in a way that means they aren't so relevant to my existing approaches.

Summary

In this chapter, we've covered everything you are likely to need when using media queries and container size and style queries for a responsive design. You now understand how to change a layout based on the characteristics of the environment your web pages find themselves in.

We've learned what media and container queries are, why we need them, and how to include them in our CSS files in all manner of ways. We've also learned how to use the `meta viewport` tag to make browsers render pages in the manner we'd like.

As I'll no doubt repeat in later chapters, don't get bogged down or intimidated by the syntax. Just get an understanding of the underlying principles and you can look up the syntax at any time; after all, that's what all developers do! As long as you understand what you can accomplish with media and container queries, our work here is largely done.

Back in *Chapter 1*, we noted that the three tenets of responsive web design are media queries, flexible layouts, and flexible media. We're three chapters in and we've only covered media queries! We're going to put that right in the next chapter.

In *Chapter 4*, we are going to take a deep dive into fluid layouts with Flexbox. We will cover how to convert a fixed-width design into a fluid proportional layout and lay out our page elements with Flexbox.

It's going to be another packed chapter, so get yourself comfy and I'll see you there!

4

Fluid Layout and Flexbox

At the end of the last chapter, we reminded ourselves that the three core tenets of responsive web design are fluid layout, media queries, and flexible media. In *Chapter 3*, we learned all about media queries and container queries. Now we know how to wield them to change a layout at a particular **breakpoint**. In this chapter, we will focus on another pillar of responsive web design: fluid layout.

Eons ago, in the mists of time (well, the late 1990s), websites were typically built with their widths defined as percentages. These percentage-based widths were fluidly adjusted to the screen and were known as fluid layouts.

In the years after, in the mid-to-late 2000s, there was an intervening fixation on fixed-width designs—I blame those pesky print designers and their obsession with pixel-perfect precision! Nowadays, as we build responsive web designs, we need to look back to fluid layouts and remember all the benefits they offer.

Until fairly recently, web developers have used several CSS layout mechanisms to create great fluid layouts. If you've worked on the web for any length of time, you will be familiar with blocks, inline blocks, tables, and other techniques to achieve any given layout. As I write this in 2025, there seems to be little benefit in giving those old techniques more than a cursory mention. We now have two powerful CSS layout mechanisms at our disposal: CSS flexbox and CSS grid.

This chapter will deal with CSS flexbox. In *Chapter 5*, *Layout with CSS Grid*, I've got a feeling you can guess what we will be covering.

Flexbox is so useful because it can do more than merely provide a fluid layout mechanism. Do you want to center content easily, change the source order of markup, and generally create amazing layouts with ease? Flexbox has you covered.

In this chapter, we will do the following:

- Learn how to convert fixed pixel layouts into proportional sizes
- Recognize flexbox as a practical path beyond these limitations
- Understand the flexible box layout module and the benefits it offers

Let's crack on with our first task: converting fixed designs into fluid relationships. This is a task that you'll need to perform constantly when building responsive web designs.

Converting a fixed pixel design into a fluid proportional layout

Graphic composites, or **comps**, as they are often called, exported from a program such as Photoshop, Illustrator, Figma, or Sketch, all have fixed pixel dimensions. At some point, the designs need to be converted into proportional dimensions when recreating the design as a fluid layout for the browser.

There is a beautifully simple formula for making this conversion that the father of responsive web design, Ethan Marcotte, set out in his 2009 article, *Fluid Grids* (`https://alistapart.com/article/FLUIDGRIDS`):

```
target / context = result
```

Put another way, divide the units of the thing you want by the thing it lives in. Let's put that into practice. Understanding it will enable you to convert any fixed-dimension layouts into responsive/fluid equivalents.

Consider a very basic page layout intended for desktop. In an ideal world, we would always be moving to a desktop layout from a smaller screen layout; however, for the sake of illustrating the proportions, we will look at the two situations back to front.

Here's an image of the layout:

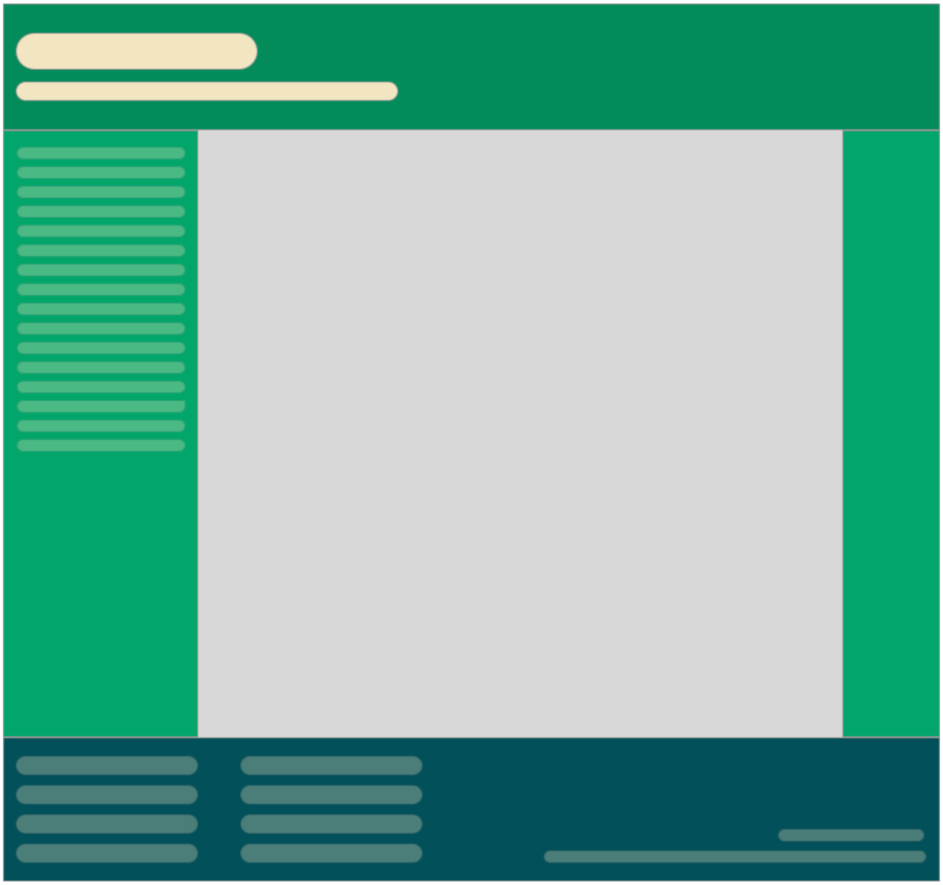


Figure 4.1: A basic "desktop" layout

The layout is 960 px wide. Both the header and footer are the full width of the layout. The left-hand area is 200 px wide, and the right-hand area is 100 px wide. That leaves 660 px for the main content area. Our job is to convert this fixed-width design into a fluid layout that retains its proportions as it is resized. For our first task, we need to convert the middle and side sections into proportional dimensions.

We will begin by converting the left-hand side. The left-hand side is 200 units wide. This value is our target value. We will divide that target size by 960 units, our context, and we have a result: 0.208333333. Now, whenever we get our result with this formula, we need to shift the decimal point two positions to the right. This gives us a value that is the target value described as a percentage of its parent. In this case, the left-hand section is 20.8333333% of its parent.

Let's practice the formula again on the middle section. Our target value is 660. Divide that by our context of 960 and we get 0.6875. Move the decimal two points to the right and we have 68.75%.

Finally, let's look at the right-hand section. Our target is 100. We divide that by the context of 960 and we get 0.104166667. Move the decimal point and we have a value of 10.4166667%.

That's as difficult as it gets. Say it with me: target divided by context equals result.

> You can use values with long decimal values with no issues in CSS. Or, if you would rather see more palatable numbers in your code, rounding them to two decimal points will work just as well for the browser.

To prove the point, let's quickly build that basic layout as blocks in the browser. To make it easier to follow along, I have added a class to the various elements that describes which piece of the **comp** they are referring to. Ordinarily, it's not a good idea to name things based on their location. The location can change, especially with a responsive design. In short, do as I say and not as I do here!

You can view the layout as `example_04-01`. Here is the HTML:

```
<div class="Wrap">
    <header class="Header"></header>
    <div class="WrapMiddle">
        <aside class="Left"></aside>
        <main class="Middle"></main>
        <aside class="Right"></aside>
    </div>
    <footer class="Footer"></footer>
</div>
```

Here is the CSS:

```
html,
body {
    margin: 0;
    padding: 0;
}
.Wrap {
    max-width: 1400px;
    margin: 0 auto;
}
```

```
.Header {
    width: 100%;
    height: 130px;
    background-color: #038c5a;
}
.WrapMiddle {
    width: 100%;
    font-size: 0;
}
.Left {
    height: 625px;
    width: 20.83%;
    background-color: #03a66a;
    display: inline-block;
}
.Middle {
    height: 625px;
    width: 68.75%;
    background-color: #bbbf90;
    display: inline-block;
}
.Right {
    height: 625px;
    width: 10.41%;
    background-color: #03a66a;
    display: inline-block;
}
.Footer {
    height: 200px;
    width: 100%;
    background-color: #025059;
}
```

If you open the example code in a browser and resize the page, you will see that the dimensions of the `.Left`, `.Middle`, and `.Right` sections remain proportional to one another. You can also play around with the `max-width` value of the `.Wrap` values to make the bounding dimensions for the layout bigger or smaller (in the example, it's set to 1,400 px).

Now, let's consider how we would have the same content on a smaller screen that flexes to a point and then changes to the layout we have already seen. You can view the final code of this layout in `example_04-02`.

The idea is that, for smaller screens, we will have a single "tube" of content. The left-hand area will only be viewable as an "off-canvas" area—typically, an area for a menu or similar, which sits off the viewable screen area and slides in when a menu button is pressed. The main content sits below the header, then the right-hand section below that, and finally the footer area. In our example, we can expose the left-hand menu area by clicking anywhere on the header. Typically, when making this kind of design pattern for real, a menu button would be used to activate the side menu.

As you would expect, when combining this with our newly mastered media query skills, we can adjust the viewport and the design just "responds"—effortlessly moving from one layout to another and stretching between the two. I'm not going to list all the CSS properties here; it's all in `example_04-02`. However, here's an example—the left-hand section:

```css
.Left {
    height: 625px;
    background-color: #03a66a;
    display: inline-block;
    position: absolute;
    left: -200px;
    width: 200px;
    font-size: 0.9rem;
    transition: transform 0.3s;
}
@media (min-width: 40rem) {
    .Left {
        width: 20.8333333%;
        left: 0;
        position: relative;
    }
}
```

You can see that, up first, without a media query, is the small screen layout. Then, at larger screen sizes, the width becomes proportional, the positioning relative, and the left value is set to 0. We don't need to rewrite properties such as `height`, `display`, or `background-color` as we aren't changing them.

This is progress. We have combined two of the core responsive web design techniques we have covered: converting fixed dimensions into proportions and using media queries to target CSS rules relevant to the viewport size.

> In a real project, we should make some provision if JavaScript isn't available and we need to view the content of the menu. We deal with this scenario in detail in *Chapter 12, Transitions, Transformations, and Animations*.

We have now covered the essentials of fluid design. To summarize, where needed, make the dimensions of elements proportional rather than fixed. This way, designs adapt to the size of their container. You now have the elements of the simple `target / context = result` formula to make the necessary calculations.

Speaking of fluidity, this seems as good a point as any to talk about flexbox—a layout system designed with fluid layouts in mind.

Why do we need flexbox?

We are now going to explore using CSS flexible box layout, or flexbox, as it is more commonly known.

In previous editions of this book, this is the point where I would have explained the shortcomings of prior CSS layout mechanisms, but as I write this in 2025, it would be like explaining why people use cars instead of horses. That said, while we won't cover them here, it is worth noting that even display types such as table have their genuine use cases, just not as a general layout approach.

Nowadays, there are two capable and powerful CSS layout systems available to us and perfectly suited to responsive design. To lay things out in one direction at a time, we can reach for Flexbox. For two directions at a time, we reach for grid, the subject of the next chapter.

Introducing flexbox

Flexbox was designed to address the shortcomings of the CSS layout mechanisms that had come before.

Here are a few choice reasons why it is so useful:

- It can easily center contents vertically
- It can change the visual order of elements
- It can automatically space and align elements within a box, automatically assigning available space between them

- Items can be laid out in a row, a row going in the reverse direction, a column down the page, or a column going in reverse order up a page
- It can make you look 10 years younger (probably not, but in low numbers of empirical tests (me), it has been proven to reduce stress compared with other layout techniques)

The bumpy path to flexbox

Flexbox has been through a few major iterations before arriving at the stable version we have today. For example, consider the changes between the 2009 version (`https://www.w3.org/TR/2009/WD-css3-flexbox-20090723/`), the 2011 version (`https://www.w3.org/TR/2011/WD-css3-flexbox-20111129/`), and the 2014 version that we are basing our examples on (`https://www.w3.org/TR/css-flexbox-1/`). The syntax differences are marked.

These differing specifications mean there have been three major implementations across browsers. Today, we need not concern ourselves with the differences. However, if you find yourself having to support some incredibly old browser (I'm talking IE11 old), be aware that you will need to ensure you have a decent autoprefixing solution as part of your build step.

Choosing your autoprefixing solution

For the sake of your sanity, if you do need to add vendor prefixes to CSS, use some form of automatic prefixing solution. There are myriad solutions these days, and many build tools, such as Vite (vitejs.dev), include them as standard.

Unless essential, there are no vendor prefixes in the code samples of this book.

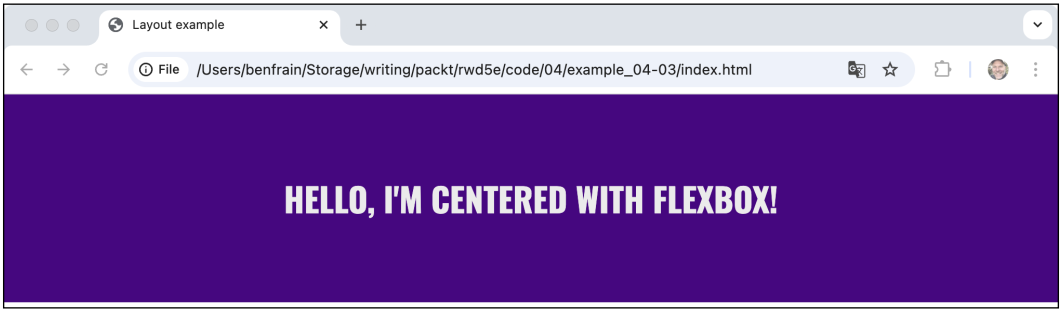
Getting flexy

Flexbox has four key characteristics: direction, alignment, ordering, and flexibility. We'll cover all of these characteristics and how they relate to each other by way of a few examples.

The examples are deliberately simplistic; we are just moving some boxes and their content around so that we can understand the principles of how flexbox works.

Perfect vertically centered text

Note that this first flexbox example is `example_04-03`.


Layout example
File /Users/benfrain/Storage/writing/packt/rwd5e/code/04/example_04-03/index.html

HELLO, I'M CENTERED WITH FLEXBOX!

Figure 4.2: Centering is simple with flexbox

Here's the markup:

```
<div class="CenterMe">Hello, I'm centered with Flexbox!</div>
```

Here is the entire CSS rule that's styling that markup:

```
.CenterMe {
    background-color: indigo;
    color: #ebebeb;
    font-family: "Oswald", sans-serif;
    font-size: 2rem;
    text-transform: uppercase;
    height: 200px;
    display: flex;
    align-items: center;
    justify-content: center;
}
```

The majority of the property/value pairs in that rule are merely setting the colors and font sizes. The three properties we are interested in are as follows:

```
.CenterMe {
    /* other declarations */
    display: flex;
    align-items: center;
    justify-content: center;
}
```

If you have not used flexbox or any of the properties in the related box alignment specification (`https://www.w3.org/TR/css-align-3/`), these properties probably seem a little alien. Let's consider what each one does:

- `display: flex`: This is the bread and butter of flexbox. This merely sets the item to be a flexbox, as opposed to a block or inline block.
- `align-items`: This aligns the items within a flexbox in the cross axis, vertically centering the text in our example.
- `justify-content`: This sets the main axis, horizontally centering the content. With a flexbox row, you can think of it as the button in a word processor that sets the text to the left, right, or center (although there are additional justify-content values that we will look at shortly).

> Flexbox has an even more convenient way of positioning items, and that's with `place-items`. But I think it is confusing and likely won't do what you hope! It provides a shorthand for setting *align items* and *justify items*. However, in my experience, I typically want to align *items* and justify *content*. So, if you are trying to use `place-items` to center your content as we have in the prior example, you will find it doesn't. To add to the confusion, in the next chapter, you will learn that you can center things with a single line with grid using, you guessed it, `place-items`. So, the takeaway here is that with flexbox, you need to use a minimum of three lines to center content, while with grid, you use just two. Also, you don't get the same result using `place-items` in flexbox as you would with grid.

OK, before we go further into the properties of flexbox, we will consider a few more examples.

> In some of these examples, I'm making use of the Google-hosted font Oswald, with a fallback to a sans serif font. In *Chapter 7*, *Web Typography*, we will look at how we can use the `@font-face` rule to link to custom font files.

Offset items

How about a simple list of navigation items, but with one offset to one side?

Here's what it looks like:

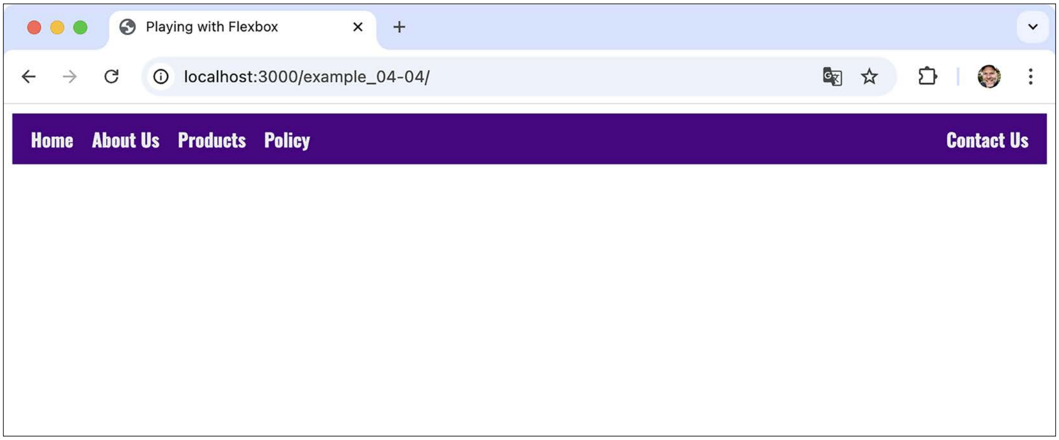

Playing with Flexbox
localhost:3000/example_04-04/
Home
About Us
Products
Policy
Contact Us

Figure 4.3: Flexbox makes it simple to offset one link in a list

You can play about with this in the `example_04-04` code. Here's the markup:

```
<div class="container">
    <a href="#" class="item">Home</a>
    <a href="#" class="item">About Us</a>
    <a href="#" class="item">Products</a>
    <a href="#" class="item">Policy</a>
    <a href="#" class="last-item">Contact Us</a>
</div>
```

Here is the CSS:

```
.container {
    background-color: indigo;
    font-family: "Oswald", sans-serif;
    font-size: 1rem;
    min-height: 2.75rem;
    display: flex;
    align-items: center;
    padding: 0 1rem;
}
.item,
.last-item {
    color: #ebebeb;
    text-decoration: none;
}
.item {
    margin-right: 1rem;
}
.last-item {
    margin-left: auto;
}
```

When you set `display: flex;` on a wrapping element, the children of that element become flex items, which then get laid out using the flex layout model. The magical property here is `margin-left: auto;`, which makes that item use all of the available margin on that side.

Reverse the order of items

Want to reverse the order of the items?

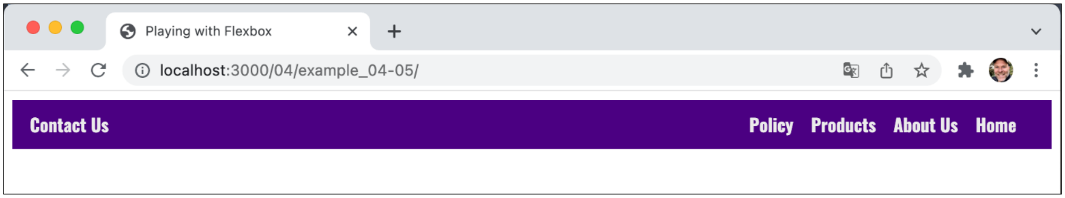

Playing with Flexbox
localhost:3000/04/example_04-05/
Contact Us
Policy
Products
About Us
Home

Figure 4.4: Reversing the visual order with Flexbox

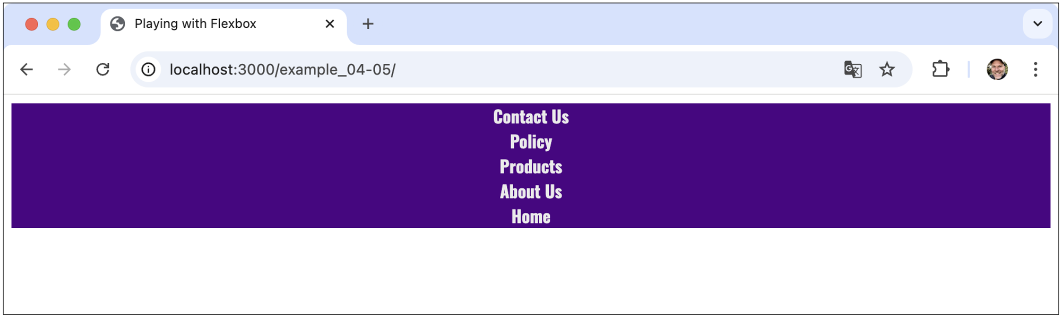
It's as easy as adding `flex-direction: row-reverse;` to the wrapping element and changing `margin-left: auto;` to `margin-right: auto;` on the offset item:

```
.container {
    background-color: indigo;
    font-family: "Oswald", sans-serif;
    font-size: 1rem;
    min-height: 2.75rem;
    display: flex;
    flex-direction: row-reverse;
    align-items: center;
    padding: 0 1rem;
}
.item,
.last-item {
    color: #ebebeb;
    text-decoration: none;
}
.item {
    margin-right: 1rem;
}
.last-item {
    margin-right: auto;
}
```

> Be aware that when using the `reverse`-based properties, you are only changing the order in presentation. This won't change the order of the content for people using a screen reader, for example. If the order of the items is essential to the understanding of the content, ensure you make the changes in the DOM to benefit all users.

How about if we want them laid out vertically instead?

Simple. Change to `flex-direction: column;` on the wrapping element and remove the auto margin:



Contact Us
Policy
Products
About Us
Home

Figure 4.5: Items laid out vertically instead

Here's the code for it, and you can play about with it using `example_04-05`:

```
.container {
    background-color: indigo;
    font-family: "Oswald", sans-serif;
    font-size: 1rem;
    min-height: 2.75rem;
    display: flex;
    flex-direction: column;
    align-items: center;
    padding: 0 1rem;
}
.item,
.last-item {
    color: #ebebeb;
    text-decoration: none;
}
```

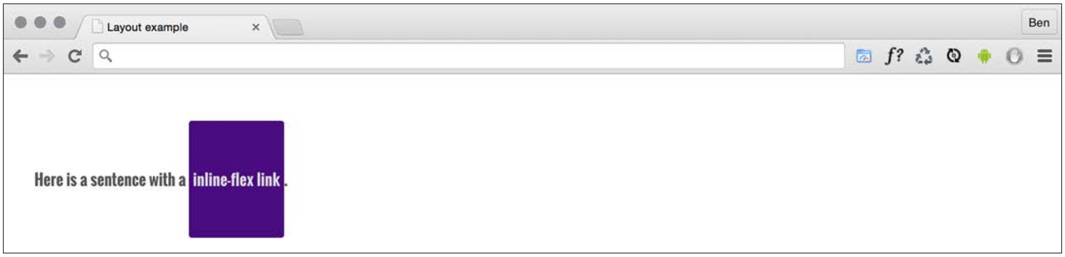
Column reverse

Want them stacked in the opposite direction? Just change to `flex-direction: column-reverse;` and you're done.

> There is a `flex-flow` property that is shorthand for setting `flex-direction` and `flex-wrap` in one go. For example, `flex-flow: row wrap;` would set the direction to a row and set wrapping on. However, at least initially, I find it easier to specify the two settings separately.

inline-flex

Flexbox has an inline variant: `display: inline-flex;`. Thanks to its beautiful centering abilities, you can do some wacky things with very little effort:



Here is a sentence with a inline-flex link.

Figure 4.6: The inline equivalent of flex is the aptly named "inline-flex"

Here's the markup:

```
<p>
    Here is a sentence with a
    <span class="InlineFlex">inline-flex</span>.
</p>
```

And, using the same basic styles as the previous examples for the fonts, font sizes, and colors, here is the CSS needed:

```
.InlineFlex {
    display: inline-flex;
    align-items: center;
    height: 120px;
    padding: 0 4px;
    background-color: indigo;
    text-decoration: none;
    border-radius: 3px;
    color: #ddd;
}
```

When items are set as `inline-flex` anonymously, which happens if their parent element is not set to `display: flex`, then they retain whitespace (which can often manifest as an unwanted gap between elements), just like `inline-block` or `inline-table` do. However, if they are within a flex container, then whitespace is removed. Of course, you don't always have to center items within a flexbox. There are a number of different options. Let's look at those now.

Flexbox alignment properties

If you want to play along with this example, start with `example_04-06s`.

The important thing to understand with Flexbox alignment is the concept of the axis. There are two axes to consider, the **main axis** and the **cross axis**. What each of these represents depends on the direction the flexbox is set to. For example, if the direction of your flexbox is set to row, which is the default, the main axis will be the horizontal axis and the cross axis will be the vertical axis.

Conversely, if your Flexbox direction is set to column, the main axis will be the vertical axis and the cross axis will be the horizontal axis. The specification (`https://www.w3.org/TR/css-flexbox-1/#justify-content-property`) provides the following illustration to aid authors:

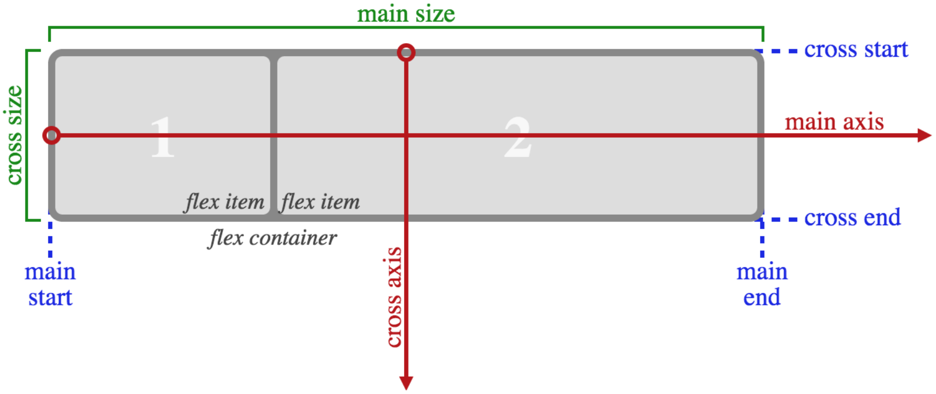

main size
cross start
cross size
1
2
main axis
flex item
flex item
cross end
flex container
main start
cross axis
main end

Figure 4.7: This image from the specification shows that the main axis always relates to the direction the flex container is heading

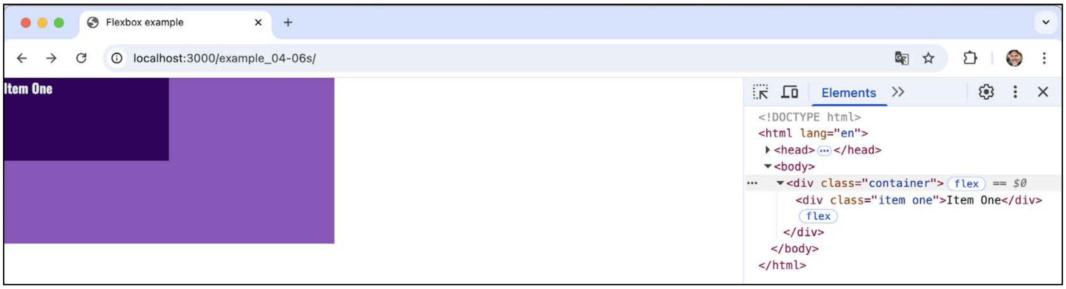
Here’s the basic markup you will want to add to the example file now if you want to try out the various alignment properties:

```
<div class="container">
    <div class="item">Item One</div>
</div>
```

Let’s set a few basic flexbox-related styles:

```
.container {
    background-color: #8f54bb;
    display: flex;
    height: 200px;
    width: 400px;
}

.item {
    background-color: #34005b;
    display: flex;
    height: 100px;
    width: 200px;
}
```

In the browser, the preceding code produces this:


Flexbox example
localhost:3000/example_04-06s/
Item One
Elements

```
<!DOCTYPE html>
<html lang="en">
 ▸<head> ⋯ </head>
 ▾<body>
···  ▾<div class="container"> flex == $0
       <div class="item one">Item One</div>
       flex
     </div>
   </body>
</html>
```

Figure 4.8: With no alignment set, child elements default to the top left

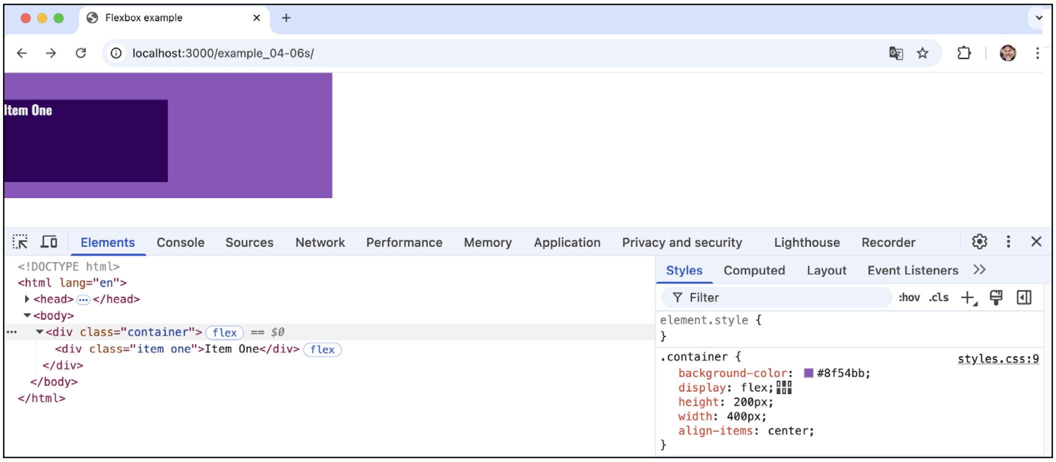
Right, let's test drive the effects of some of these properties.

align-items

The `align-items` property positions items in the cross-axis. Let's apply this property to our wrapping element, like so:

```
.container {
    background-color: #8f54bb;
    display: flex;
    height: 200px;
    width: 400px;
    align-items: center;
}
```

As you would imagine, the item within that box gets centered vertically:


Flexbox example
localhost:3000/example_04-06s/
Item One
Elements
Console
Sources
Network
Performance
Memory
Application
Privacy and security
Lighthouse
Recorder
Styles
Computed
Layout
Event Listeners
Filter
:hov
.cls

```
<!DOCTYPE html>
<html lang="en">
 ▸<head> ⋯ </head>
 ▾<body>
···  ▾<div class="container"> flex == $0
       <div class="item one">Item One</div> flex
     </div>
   </body>
</html>
```

```
element.style {
}
.container {                                styles.css:9
   background-color: ■ #8f54bb;
   display: flex;
   height: 200px;
   width: 400px;
   align-items: center;
}
```

Figure 4.9: A one-liner provides cross-axis centering

The same effect would be applied to any number of children within.

align-self

Sometimes, you may want to pull just one item into a different alignment. Individual flex items can use the `align-self` property to align themselves.

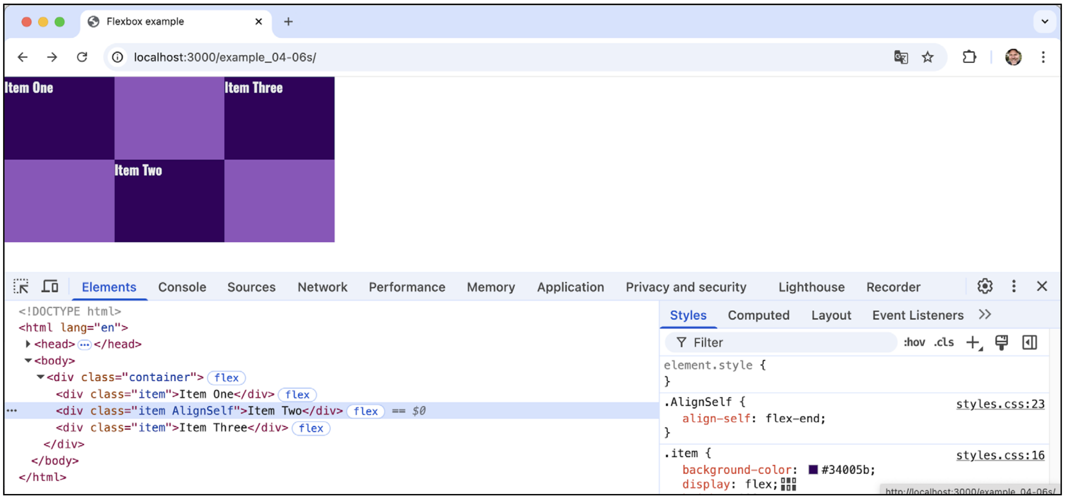
At this point, let's *remove the previous alignment properties* in the CSS and also add another two `div` elements to the markup, both also with a class of `item`. In the middle of these three items, we will add an additional class of `AlignSelf`. We'll use that class in the CSS to add the `align-self` property. So, here's the HTML:

```
<div class="container">
    <div class="item">Item One</div>
    <div class="item AlignSelf">Item Two</div>
    <div class="item">Item Three</div>
</div>
```

Here is the CSS for that one item we want to move:

```
.AlignSelf {
    align-self: flex-end;
}
```

Here is the effect in the browser:


Flexbox example
localhost:3000/example_04-06s/
Item One
Item Three
Item Two
Elements
Console
Sources
Network
Performance
Memory
Application
Privacy and security
Lighthouse
Recorder
Styles
Computed
Layout
Event Listeners
Filter
:hov
.cls

```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
 ▼<body>
   ▼<div class="container"> flex
      <div class="item">Item One</div> flex
      <div class="item AlignSelf">Item Two</div> flex == $0
      <div class="item">Item Three</div> flex
    </div>
  </body>
</html>
```

```
element.style {
}
.AlignSelf {                              styles.css:23
  align-self: flex-end;
}
.item {                                   styles.css:16
  background-color: #34005b;
  display: flex;
```

Figure 4.10: Individual items can be aligned in a different manner

Wow! Flexbox really makes these kinds of changes trivial. In the example, the value of `align-self` was set to `flex-end`. Let's consider the possible values we could use on the cross axis before looking at alignment in the main axis.

Possible alignment values

For cross-axis alignment, flexbox has the following possible values:

- `flex-start`: Setting an element to `flex-start` makes it begin at the "starting" edge of its flex container
- `flex-end`: Setting to `flex-end` aligns the element at the end of the flex container
- `center`: This puts it in the middle of the flex container
- `baseline`: This sets all the flex items in the container so that their baselines align
- `stretch`: This makes the items stretch to the size of their flex container (in the cross-axis)

There are some particulars inherent to using these properties, so if something isn't playing happily, always refer to the specification for any edge case scenarios: `https://www.w3.org/TR/css-flexbox-1/#align-items-property`.

justify-content

Alignment in the main axis is controlled with `justify-content`. Possible values for `justify-content` are the following:

- `flex-start`
- `flex-end`
- `center`
- `space-between`
- `space-around`
- `space-evenly`

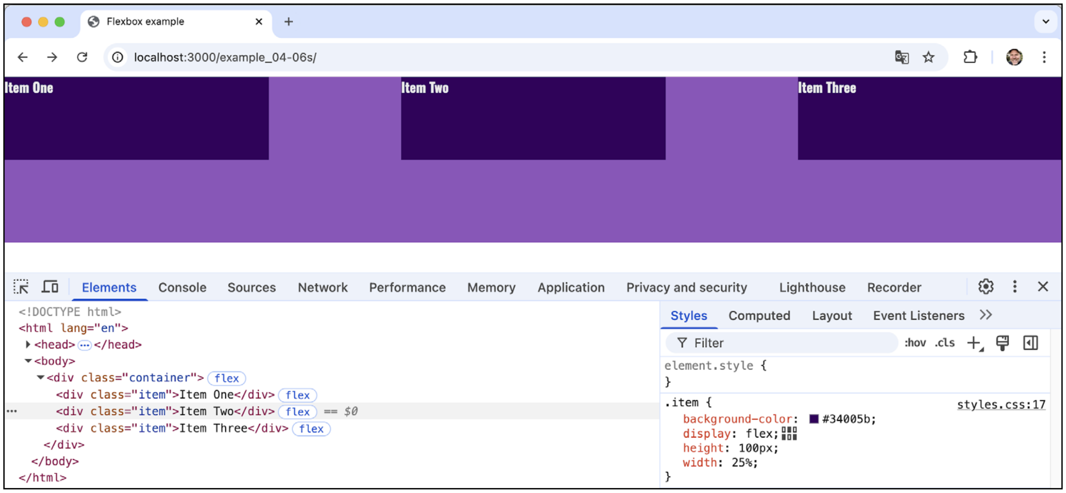
The first three do exactly what you would now expect. However, let's take a look at what the last three do. Adjust the markup in the example, or simply imagine that the markup is back to this:

```html
<div class="container">
    <div class="item">Item One</div>
    <div class="item">Item Two</div>
    <div class="item">Item Three</div>
</div>
```

Then consider the following CSS. We are setting the three `div` elements with a class of `item` to each be 25% width, wrapped by a flex container, with a class of `container`, set to be 100% width:

```css
.container {
    background-color: #8f54bb;
    display: flex;
    justify-content: space-between;
    height: 200px;
    width: 100%;
}
.item {
    background-color: #34005b;
    display: flex;
    height: 100px;
    width: 25%;
}
```

As the three items will only take up 75% of the available space, `justify-content` explains what we would like the browser to do with the remaining space. A value of `space-between` puts an equal amount of space between the items, and `space-around` puts it around the items. Perhaps a screenshot or two will help here —this is `space-between`:

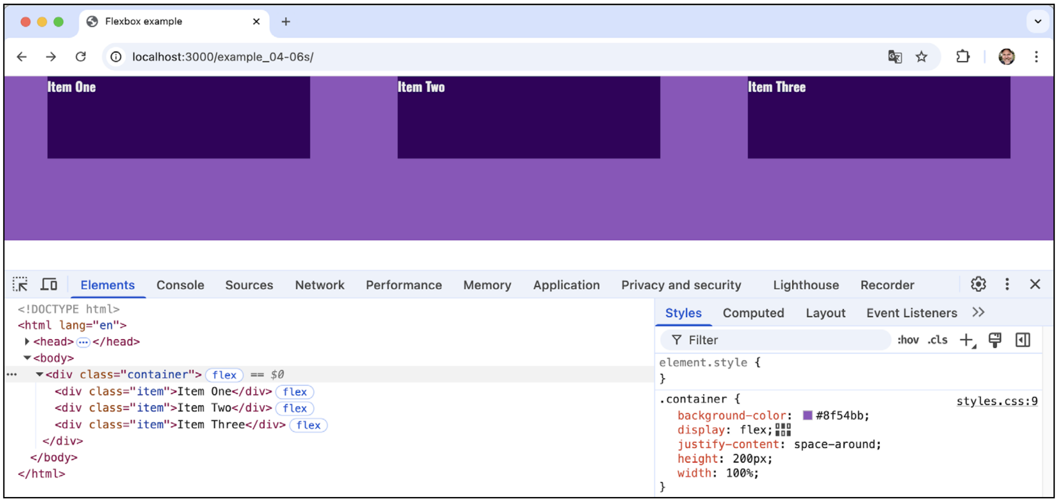

Flexbox example
localhost:3000/example_04-06s/
Item One
Item Two
Item Three
Elements
Console
Sources
Network
Performance
Memory
Application
Privacy and security
Lighthouse
Recorder
Styles
Computed
Layout
Event Listeners
Filter
:hov
.cls

```
<!DOCTYPE html>
<html lang="en">
 ▸<head> ··· </head>
 ▾<body>
   ▾<div class="container"> flex
      <div class="item">Item One</div> flex
      <div class="item">Item Two</div> flex == $0
      <div class="item">Item Three</div> flex
    </div>
  </body>
</html>
```

```
element.style {
}
.item {                                   styles.css:17
  background-color: ■ #34005b;
  display: flex;
  height: 100px;
  width: 25%;
}
```

Figure 4.11: Main axis alignment is carried out with the justify-content property

Here is what happens if we switch to `space-around`:

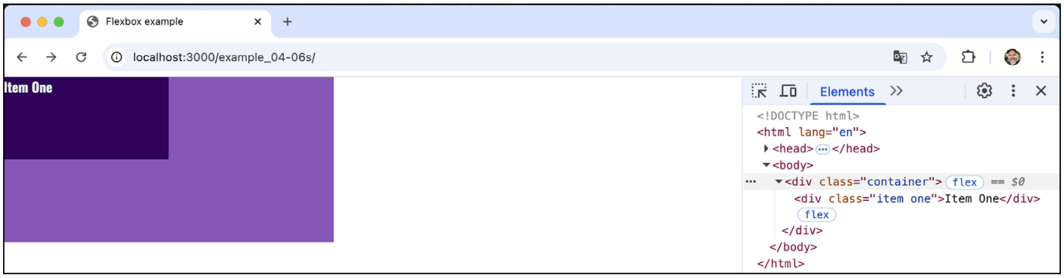

Flexbox example
localhost:3000/example_04-06s/
Item One
Item Two
Item Three
Elements
Console
Sources
Network
Performance
Memory
Application
Privacy and security
Lighthouse
Recorder
Styles
Computed
Layout
Event Listeners
Filter
:hov .cls
styles.css:9

```
<!DOCTYPE html>
<html lang="en">
 ▸<head> ⋯ </head>
 ▾<body>
   ▾<div class="container"> flex == $0
      <div class="item">Item One</div> flex
      <div class="item">Item Two</div> flex
      <div class="item">Item Three</div> flex
    </div>
  </body>
</html>
```

```
element.style {
}
.container {
  background-color: #8f54bb;
  display: flex;
  justify-content: space-around;
  height: 200px;
  width: 100%;
}
```

Figure 4.12: Subtly different, but notice the space around, not just between, items

Finally, let's look at what `space-evenly` does. This takes the available space and adds an equal amount to every gap.


Flexbox example
localhost:3000/example_04-06s/
Item One
Elements

```
<!DOCTYPE html>
<html lang="en">
 ▸<head> ⋯ </head>
 ▾<body>
   ▾<div class="container"> flex == $0
      <div class="item one">Item One</div>
      flex
    </div>
  </body>
</html>
```

Figure 4.13: An even space between items with space-evenly

In those prior examples, the difference between `space-evenly` and `space-around` may not be obvious. When you use `space-evenly`, the space between each item and the space at the beginning and end of all the items is the same. With `space-around`, only the space between the items is the same; the space at the beginning and end is the same as the space that would be added on one side of an item (so half the distance between the items).

It is also worth pointing out that when it comes to justify properties, you can also use the `left` and `right` keywords. Those keywords make sense in a typical flex direction, but make my head hurt when the direction is set to column, so I personally avoid using them.

The various alignment properties of flexbox are currently specified in CSS box alignment module level 3. This should give the same fundamental alignment powers to other display properties, such as `display: block;` and `display: table`. The specification is still being worked on, so you can keep checking the status at `https://www.w3.org/TR/css-align-3/`.

The flex property

We've used the `width` property on those flex items, but it's also possible and preferable to define the width, or "flexiness," if you will, with the `flex` property. To illustrate, consider an example with the same markup as the last, but with amended CSS for the items:

```
.item {
    background-color: #34005b;
    display: flex;
    height: 100px;
    flex: 1;
    border: 1px solid #ebebeb;
}
```

Here is what we get in the browser:

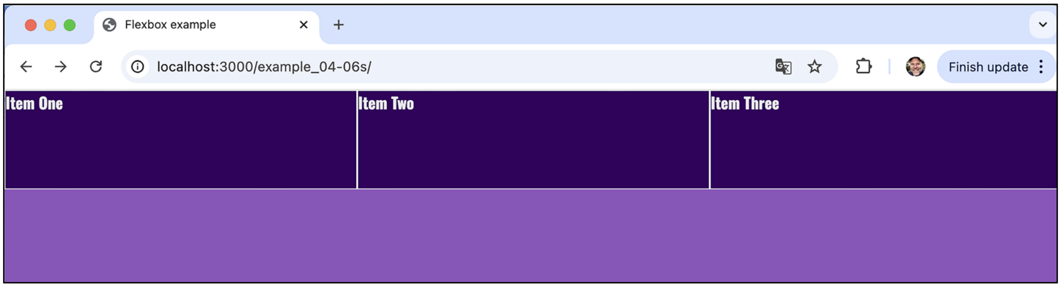


Figure 4.14: Our first foray into using the flex shorthand

Those three items have gobbled up all the available space. So, what's going on?

The `flex` property is actually a shorthand way of specifying three separate properties: `flex-grow`, `flex-shrink`, and `flex-basis`. The specification covers these individual properties in more detail here: `https://www.w3.org/TR/css-flexbox-1/#flex-components`. However, the specification recommends that authors use the `flex` shorthand property, so that's what we will be learning here, capiche?

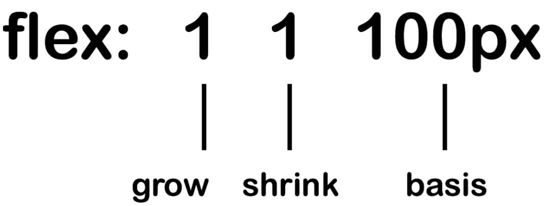


Figure 4.15: Understanding the three possible values of the flex property

For items within a flex container, if a `flex` property is present, it is used to size the item rather than a `width` or `height` value (if also present). Even if the `width` or `height` value is specified after the `flex` property, it will still have no effect.

> However, it is important to note that if the item you are adding the `flex` property to is not a flex item, the `flex` property will have no effect.

Now, let's look at what each of these flex properties actually does:

- `flex-grow` is the first value you can pass to `flex`. It is the amount, in relation to any sibling flex items, that the flex item can grow when free space is available.
- `flex-shrink` is the second value you can pass. It is the amount the flex item can shrink, in relation to the other sibling flex items, when there is not enough space available.
- `flex-basis` is the final value you can pass to `flex`. It is the base size that the flex item is set to. If you don't set an item to shrink, you can use `flex-basis` like `min-width`.

Although it's possible to just write `flex: 1` and have that interpreted to mean `flex: 1 1 0`, I recommend writing all the values into a `flex` shorthand property yourself. I think it's clearer to understand what you intend to happen that way.

For example, `flex: 1 1 auto` means that the item will grow into one part of the available space. It will also shrink one part when space is lacking, and the basis size for the flexing is the intrinsic width of the content, that is, the size the content would be if `flex` weren't involved.

Let's try another: `flex: 0 0 50px` means this item will neither grow nor shrink, and its basis is 50 px (so it will be 50 px regardless of any free space).

What about `flex: 2 0 50%`? That's going to take two "lots" of available space; it won't shrink, and its basis size is 50%. What do you think will happen if you have three items all set to that width in a fixed-width container? Well, let's think this through.

Our shorthand has `flex-basis` set to 50%, so each item is going to start out at 50% of the container. As there are three of them, one is going to overflow. Now, we don't have `flex-shrink` set, so none of them are going to get any smaller. Although we have `flex-grow` set to 2, there is no room to grow, so we get this:

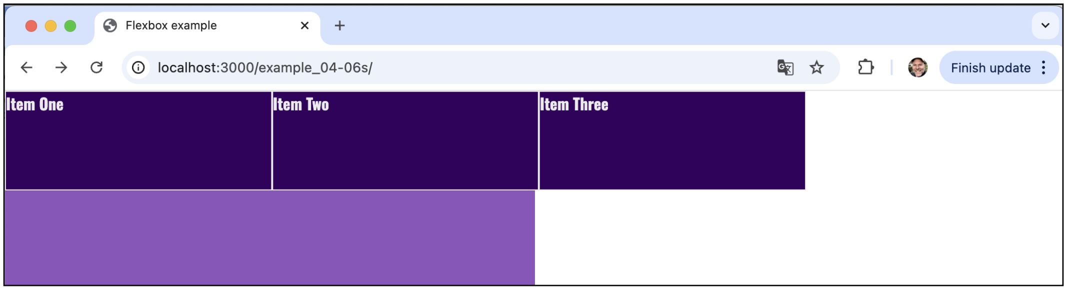


Figure 4.16: If items don't fit, like anything else, they overflow, wrap, or scroll if you set them to do so

Hopefully, these brief examples have demystified the `flex` property a little, and this incredibly powerful property is starting to make a little sense. If you take a look at `example_04-06`, the first item is set to `flex: 1.5 0 auto;`. Hopefully, you can imagine what that will look like before opening it in the browser to find out.

I could write chapters and chapters on flexbox! There are so many examples we could look at.

However, for now, there are just two more things I would like to share with you.

Simple sticky footer

Suppose you want a footer to sit at the bottom of the viewport when there is not enough content to push it there. Historically, this has been a pain to achieve, but with flexbox, it's simple. Consider this markup, which is `example_04-07`:

```
<body>
    <main>
        Here is a bunch of text but not enough to push the footer to
        the bottom of the page.
    </main>
    <footer>
        I'd like this to be at the bottom and thanks to Flexbox
        I can be put in my place!
    </footer>
</body>
```

Here's the CSS:

```
html,
body {
    margin: 0;
    padding: 0;
    font-family: "Oswald", sans-serif;
    color: #ebebeb;
}

html {
    height: 100%;
}

body {
    background-color: #ebebeb;
    display: flex;
    flex-direction: column;
    min-height: 100%;
}

main {
    flex: 1 0 auto;
    color: #333;
    padding: 1.5rem;
}

footer {
    background-color: violet;
    color: #fff;
    padding: 1.5rem;
}
```

Take a look at that in the browser, and test it by adding more content into `main`. You'll see that when there is not enough content, the footer is stuck to the bottom of the viewport. When there is enough, it sits below the content.

This works because our `flex` property is set to grow where space is available. As our `body` is a flex container of 100% minimum height, the main content can grow into all of that available space. Beautiful!

Visually re-ordering items

Flexbox has the ability to visually reorder items. Prior to flexbox, if you wanted to have something that came after something else in the DOM appear before it instead, you were in for a rough time. However, flexbox makes such work trivial. Let's have a look at how it works.

Consider this markup, which you can build along with in `example_04-08`:

```
<div class="container">
    <header class="item">header</header>
    <aside class="item">aside one</aside>
    <main class="item">main</main>
    <aside class="item aside-two">aside two</aside>
    <footer class="item">footer</footer>
</div>
```

We will add some simple colors to more easily differentiate the sections, and stack them one under another in the same order they appear in the markup:

```
.container {
    background-color: indigo;
    display: flex;
    flex-direction: column;
}
.item {
    display: flex;
    align-items: center;
    min-height: 6.25rem;
    padding: 1rem;
}
header {
    background-color: #105b63;
}
main {
    background-color: #fffad5;
}
```

```
aside {
    background-color: #ffd34e;
}
aside.aside-two {
    background-color: #db9e36;
}
footer {
    background-color: #bd4932;
}
```

That renders in the browser like this:

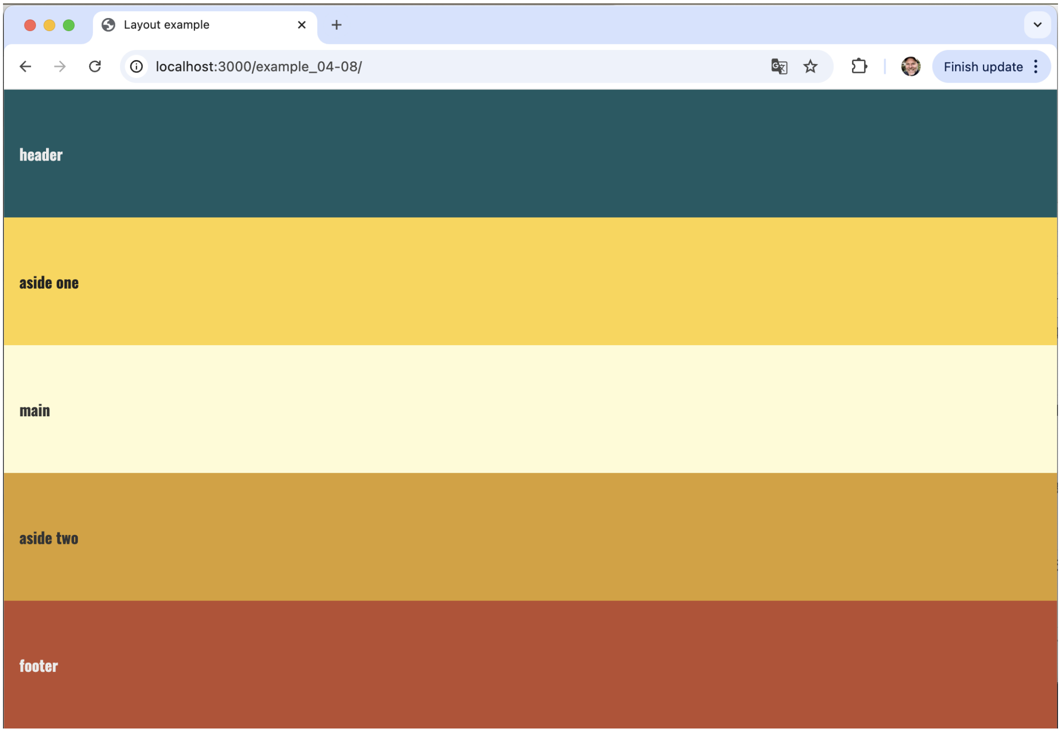

Layout example
localhost:3000/example_04-08/
Finish update
header
aside one
main
aside two
footer

Figure 4.17: Here, our boxes are displayed in source order

Now, suppose we want to switch the order of `main` to be the first item, visually, without touching the markup. With flexbox, it's as simple as adding a single property/value pair:

```
main {
    order: -1;
}
```

The `order` property lets us revise the order of items within a flexbox simply and sanely. In this example, a value of `-1` means that we want it to be before all the others.

> Remember, if you use `order` to change the visual ordering, users may jump around the page in an odd way if using *Tab* or assistive technology to move about your page. Remember, the underlying page has not changed, and so jumping to another section may be quite disconcerting. I'm not here to tell you not to use `order`, but I am here to tell you to use it responsibly. If you can have your layout match the way you are presenting it, that is almost always the better option.

If you want to switch items around quite a bit, I'd recommend being a little more declarative and adding an order number for each item. This makes things a little easier to understand when you combine them with media queries.

Let's combine our new source order-changing powers with some media queries to produce not just a different layout at different screen widths but different ordering.

Let's suppose we want our main content to be at the beginning of a document. Let's amend the content of our earlier example:

```
<div class="container">
    <main class="item">main</main>
    <aside class="item">aside one</aside>
    <aside class="item aside-two">aside two</aside>
    <header class="item">header</header>
    <footer class="item">footer</footer>
</div>
```

First is the page content, then our two sidebar areas, then the header, and finally, the footer. As we'll be using flexbox, we can structure the HTML in the order that makes sense for the document, regardless of how things need to be laid out visually.

After some basic styling for each flex item, for the smallest screens (outside of any media query), I'll go with the following order:

```
header {
    background-color: #105b63;
    order: 1;
}
main {
    background-color: #fffad5;
    color: #333;
    order: 2;
}
aside {
    background-color: #ffd34e;
    color: #222;
    order: 3;
}
aside.aside-two {
    background-color: #db9e36;
    color: #333;
    order: 4;
}
footer {
    background-color: #bd4932;
    order: 5;
}
```

That gives us this in the browser:

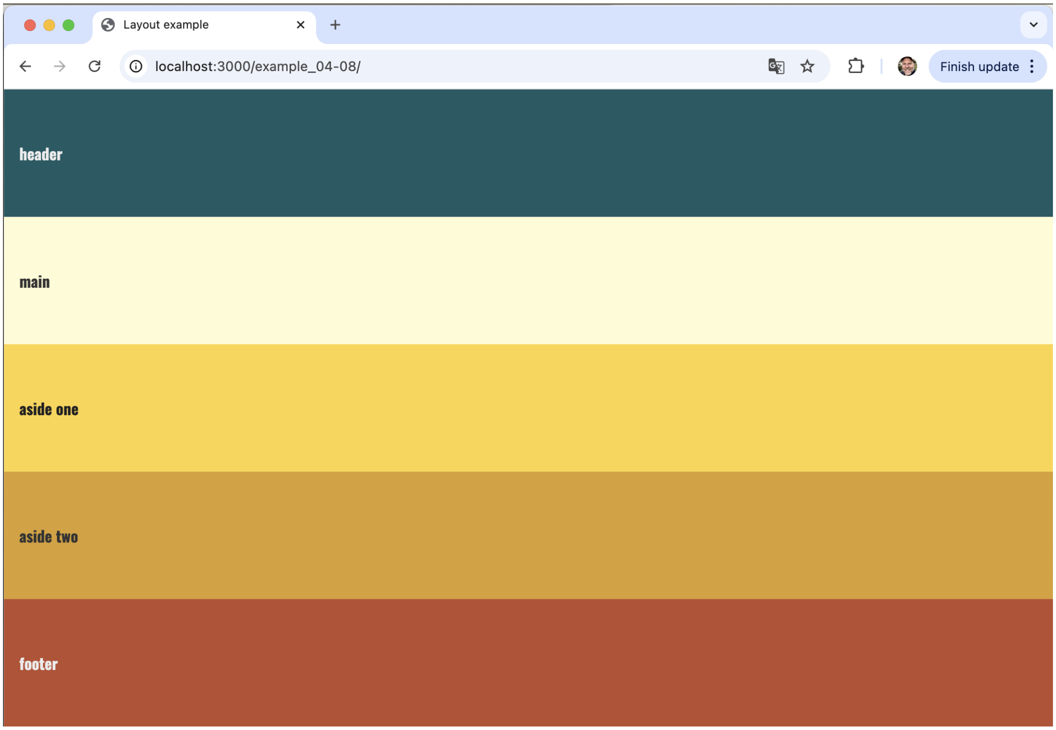

Layout example
localhost:3000/example_04-08/
Finish update
header
main
aside one
aside two
footer

Figure 4.18: We can move the items in visual order with a single property

Then, at a breakpoint, I'm switching to this:

```
@media (min-width: 600px) {
    .container {
        flex-flow: row wrap;
    }
    header {
        flex: 0 0 100%;
    }
    main {
        flex: 1 0 auto;
        order: 3;
    }
    aside {
        flex: 0 0 150px;
        order: 2;
    }
    aside.aside-two {
        flex: 0 0 150px;
        order: 4;
    }
    footer {
        flex: 1 0 100%;
    }
}
```

That gives us this in the browser:

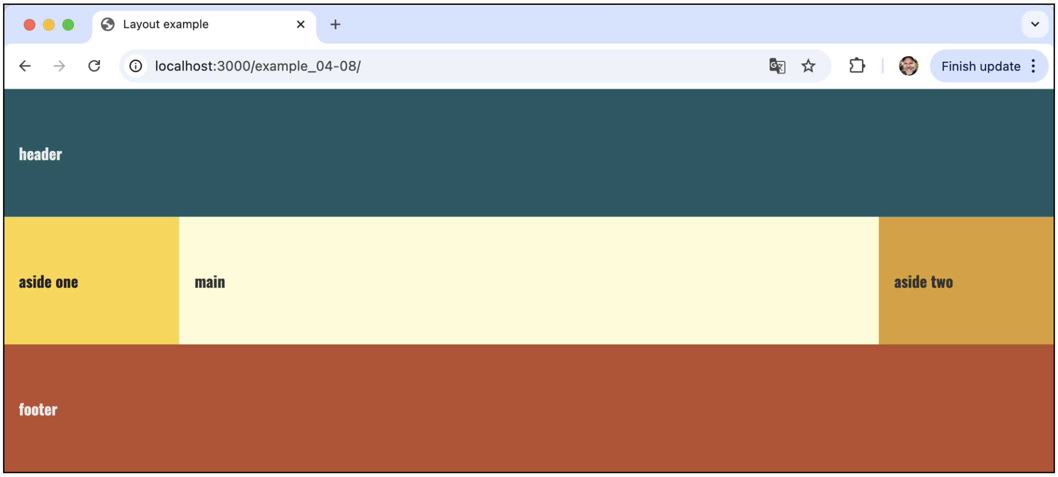

Layout example
localhost:3000/example_04-08/
Finish update
header
aside one
main
aside two
footer

Figure 4.19: With a media query, we can change the visual order again

In this example, the shortcut `flex-flow: row wrap` has been used. `flex-flow` is actually a shorthand property of sorts that lets you set two properties in one: `flex-direction` and `flex-wrap`.

We've used `flex-direction` already to switch between rows and columns and to reverse elements. However, we haven't looked at `flex-wrap` yet.

Wrapping with flex

By default, items in a flex container will shrink to fit and, if they can't, they will overflow the container. For example, consider this markup:

```
<div class="container">
    <div class="item">Item 1</div>
    <div class="item">Item 2</div>
    <div class="item">Item 3</div>
    <div class="item">Item 4</div>
</div>
```

Then consider this CSS:

```
.container {
    display: flex;
    width: 500px;
    background-color: #bbb;
    align-items: center;
    border: 1px solid #111;
}
.item {
    color: #111;
    display: inline-flex;
    align-items: center;
    justify-content: center;
    font-size: 23px;
    flex: 0 0 160px;
    height: 40px;
    border: 1px dashed #545454;
}
```

You may be wondering why the outer container is set with `width` and not the `flex` shorthand property we looked at before. Remember, this is because unless the element is a flex item (inside a flexbox itself), `flex` has no effect.

Because there is a width of only 500 px on the flex container, those four elements don't fit:

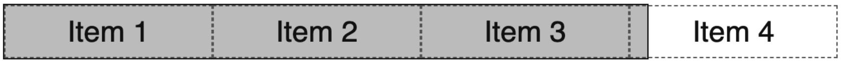


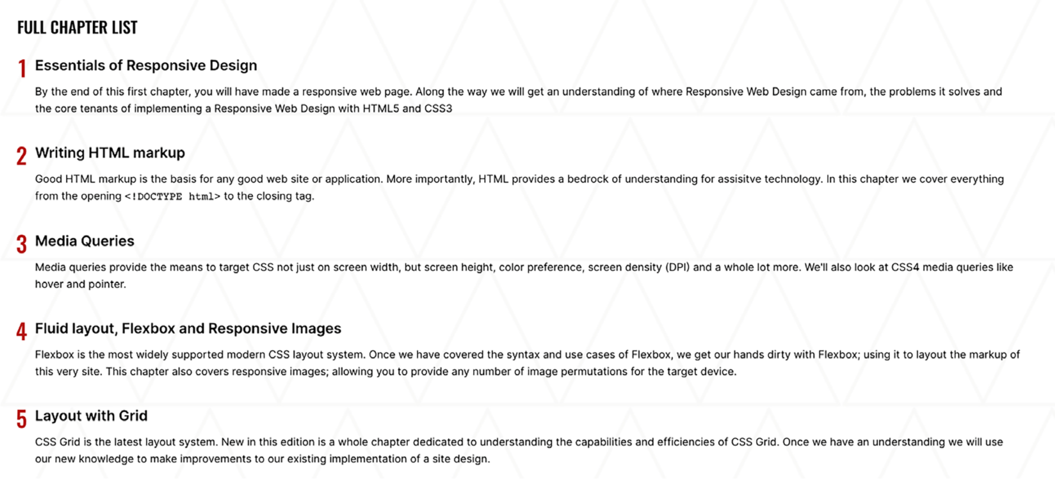
Figure 4.20: By default, flexbox will always keep child elements from wrapping

However, those items can be set to wrap with `flex-wrap: wrap`. This wraps the items once they hit the edge of the container.

The likelihood is that there will be times when you want flex items to wrap and other times when you don't. Remember that the default is not to wrap, but you can easily make the change with a single line.

Also, remember that you can set the wrapping by itself with `flex-wrap`, or as part of the `flex-flow` direction and wrap shorthand.

Let’s solve a real-world problem with `flex-wrap`. Consider the list of paragraphs in the following figure. At this width, they are not easy to read:



FULL CHAPTER LIST

1 **Essentials of Responsive Design**

By the end of this first chapter, you will have made a responsive web page. Along the way we will get an understanding of where Responsive Web Design came from, the problems it solves and the core tenants of implementing a Responsive Web Design with HTML5 and CSS3

2 **Writing HTML markup**

Good HTML markup is the basis for any good web site or application. More importantly, HTML provides a bedrock of understanding for assisitve technology. In this chapter we cover everything from the opening `<!DOCTYPE html>` to the closing tag.

3 **Media Queries**

Media queries provide the means to target CSS not just on screen width, but screen height, color preference, screen density (DPI) and a whole lot more. We'll also look at CSS4 media queries like hover and pointer.

4 **Fluid layout, Flexbox and Responsive Images**

Flexbox is the most widely supported modern CSS layout system. Once we have covered the syntax and use cases of Flexbox, we get our hands dirty with Flexbox; using it to layout the markup of this very site. This chapter also covers responsive images; allowing you to provide any number of image permutations for the target device.

5 **Layout with Grid**

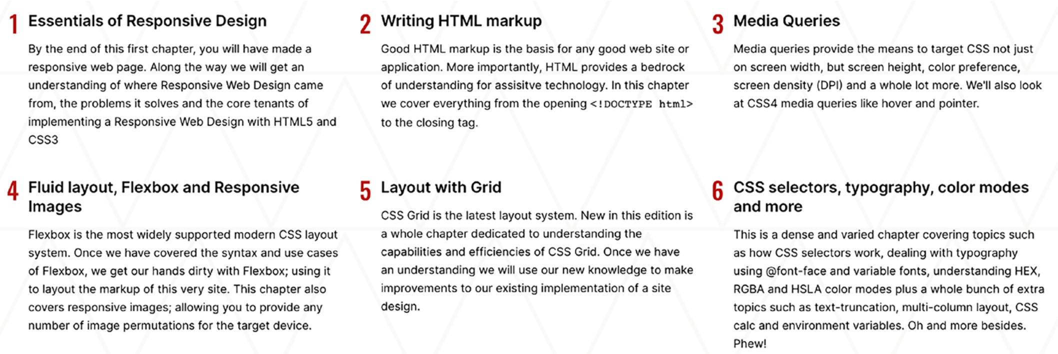
CSS Grid is the latest layout system. New in this edition is a whole chapter dedicated to understanding the capabilities and efficiencies of CSS Grid. Once we have an understanding we will use our new knowledge to make improvements to our existing implementation of a site design.

Figure 4.21: This content is overstretched at this viewport width

Let’s amend that layout with a media query and a few choice flex properties. I’m consolidating all of these changes into one media query here for brevity, but remember that you can use as many as you like to organize your code however you see fit. We covered the options extensively in *Chapter 3*, *Media and Container Queries*:

```
@media (min-width: 1000px) {
    .rwd-Chapters_List {
        display: flex;
        flex-wrap: wrap;
    }
    .rwd-Chapter {
        flex: 0 0 33.33%;
        padding: 0 20px;
    }
    .rwd-Chapter::before {
        left: -20px;
    }
}
```

That produces this effect in the browser:



1 **Essentials of Responsive Design**

By the end of this first chapter, you will have made a responsive web page. Along the way we will get an understanding of where Responsive Web Design came from, the problems it solves and the core tenants of implementing a Responsive Web Design with HTML5 and CSS3

2 **Writing HTML markup**

Good HTML markup is the basis for any good web site or application. More importantly, HTML provides a bedrock of understanding for assisitve technology. In this chapter we cover everything from the opening `<!DOCTYPE html>` to the closing tag.

3 **Media Queries**

Media queries provide the means to target CSS not just on screen width, but screen height, color preference, screen density (DPI) and a whole lot more. We'll also look at CSS4 media queries like hover and pointer.

4 **Fluid layout, Flexbox and Responsive Images**

Flexbox is the most widely supported modern CSS layout system. Once we have covered the syntax and use cases of Flexbox, we get our hands dirty with Flexbox; using it to layout the markup of this very site. This chapter also covers responsive images; allowing you to provide any number of image permutations for the target device.

5 **Layout with Grid**

CSS Grid is the latest layout system. New in this edition is a whole chapter dedicated to understanding the capabilities and efficiencies of CSS Grid. Once we have an understanding we will use our new knowledge to make improvements to our existing implementation of a site design.

6 **CSS selectors, typography, color modes and more**

This is a dense and varied chapter covering topics such as how CSS selectors work, dealing with typography using @font-face and variable fonts, understanding HEX, RGBA and HSLA color modes plus a whole bunch of extra topics such as text-truncation, multi-column layout, CSS calc and environment variables. Oh and more besides. Phew!

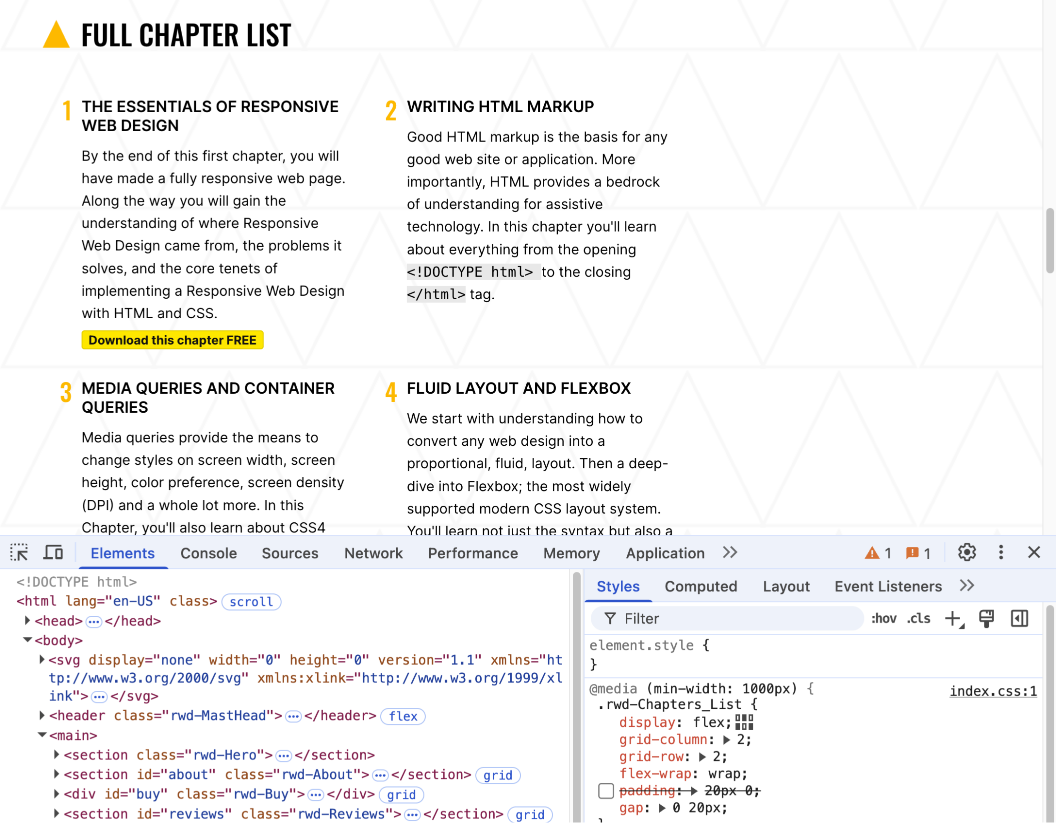
Figure 4.22: Setting the flex container to wrap means the content can space out equally

We made the container of the chapters into a flex container. Then, to stop the elements from scrunching up into one another, we set the container to wrap. To limit the chapter section widths to a third of the container, we used the flex shorthand to set 33.33% as `flex-basis` and prevented the element from growing or shrinking. Padding was used to provide a little space between them. The final small tweak was to bring in the chapter numbers a little.

What about gap?

You may have already come across the gap property if you have already used CSS grid. If not, with flex, it is a property that lets you set the gap between sibling items inside a flex container. That's great because if you want space between items, and you use something such as a margin or padding, you often have to remember to remove or reset it on the final item. gap negates that problem.

That sounds very useful. Given that, why have I not just used the gap property between the preceding items, rather than adding 20 px padding on either side? The problem here is that when flex items have their flex-basis specified with percentages, they are not able to take the gap into account, and the percentages are resolved against the size of the container. So, had we used gap instead of padding, the items would have wrapped too soon, leaving only two items on each row.



FULL CHAPTER LIST

1 THE ESSENTIALS OF RESPONSIVE WEB DESIGN

By the end of this first chapter, you will have made a fully responsive web page. Along the way you will gain the understanding of where Responsive Web Design came from, the problems it solves, and the core tenets of implementing a Responsive Web Design with HTML and CSS.

Download this chapter FREE

2 WRITING HTML MARKUP

Good HTML markup is the basis for any good web site or application. More importantly, HTML provides a bedrock of understanding for assistive technology. In this chapter you'll learn about everything from the opening `<!DOCTYPE html>` to the closing `</html>` tag.

3 MEDIA QUERIES AND CONTAINER QUERIES

Media queries provide the means to change styles on screen width, screen height, color preference, screen density (DPI) and a whole lot more. In this Chapter, you'll also learn about CSS4

4 FLUID LAYOUT AND FLEXBOX

We start with understanding how to convert any web design into a proportional, fluid, layout. Then a deep-dive into Flexbox; the most widely supported modern CSS layout system. You'll learn not just the syntax but also a

```
<!DOCTYPE html>
<html lang="en-US" class> scroll
 ▸<head> ··· </head>
 ▾<body>
   ▸<svg display="none" width="0" height="0" version="1.1" xmlns="ht
   tp://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xl
   ink"> ··· </svg>
   ▸<header class="rwd-MastHead"> ··· </header> flex
   ▾<main>
     ▸<section class="rwd-Hero"> ··· </section>
     ▸<section id="about" class="rwd-About"> ··· </section> grid
     ▸<div id="buy" class="rwd-Buy"> ··· </div> grid
     ▸<section id="reviews" class="rwd-Reviews"> ··· </section> grid
```

```
element.style {
}
@media (min-width: 1000px) {                              index.css:1
 .rwd-Chapters_List {
    display: flex;
    grid-column: ▸ 2;
    grid-row: ▸ 2;
    flex-wrap: wrap;
 ☐  ~~padding: ▸ 20px 0;~~
    gap: ▸ 0 20px;
```

Figure 4.23: gap is great unless you are dealing with percentages in the flex property

Wrapping up flexbox

There are near-endless possibilities when using the flexbox layout system and, due to its inherent “flex-ability,” it’s a perfect match for responsive design. If you’ve never built anything with flexbox before, all the new properties and values can seem a little odd, and it’s sometimes disconcertingly easy to achieve layouts that have previously taken far more work.

Summary

We've covered a lot of ground in this chapter. We started by understanding how to create fluid layouts that can flex between the media queries we set. We then spent considerable time getting acquainted with flexbox, learning how to solve common layout problems with relative ease. We learned all about how to easily align content in containers, how to visually re-order things, and how to define the way we want our content to grow and shrink within its context.

As luck would have it, two great layout mechanisms arrived in CSS in relatively quick succession. Hot on the heels of the flexible box layout module was grid layout module level 1: `https://www.w3.org/TR/css3-grid-layout/`. Like flexbox, CSS grid means learning quite a bit of unique syntax. But don't let that put you off. The next chapter is completely dedicated to grid: what it can do, how it does it, and how we can bend it to our will.

5

Layout with CSS Grid

Without a doubt, the biggest game changer to CSS layout has been Grid.

CSS Grid is capable of achieving everything that's possible with other layout methods, with less code, and more predictability, as well as enabling things we simply couldn't do before!

I'd go as far as saying Grid is revolutionary rather than evolutionary. There are entirely new concepts to understand that have no real forebears in past CSS. As such, if you are familiar with other CSS layout methods, but not Grid, expect to take some time getting comfortable using it.

But trust me, it is worth it!

In this chapter, we will learn about the following:

- What CSS Grid is and the problems it solves
- The essential concepts to understand when dealing with Grid Layout
- Grid-specific terminology
- How to set up a grid
- How to position items in a grid
- How to create powerful responsive patterns with minimal code
- How to understand and write the `grid` shorthand syntax
- The latest Grid addition, `subgrid`

Toward the end of the chapter, you will also be tasked with a small exercise to use some of the techniques we have learned to refactor the layout of a portion of the `https://rwd.education` website we have been looking at in previous chapters.

Before I ask you to do that, I'd better get on with explaining what Grid is all about.

What CSS Grid is and the problems it solves

CSS Grid is a two-dimensional layout system. Flexbox, which we covered in the last chapter, concerns itself with items being laid out in a single dimension/direction at a time. A Flexbox container either lays things out in a row, or it lays them out in a column. It cannot facilitate laying out items across and down at once; that is what Grid is for.

I should point out at the outset that you don't need to choose between Flexbox and Grid when building projects. They are not mutually exclusive. I commonly use both, even within a single visual component.

When you adopt Grid, you don't have to forsake any other display methods. Part of your interface coded with Grid can quite happily live inside a Flexbox, standard block, or `inline-block`.

So you should appreciate that there are times when using Grid is the most appropriate option, and there are times when Flexbox, or another layout method, is the most appropriate.

The truth is, we've been creating grid-like layouts with CSS for years. We just haven't had a specific `grid` layout mechanism in CSS to do it. We have made use of blocks, floats, tables, and many other ingenious techniques to work around the fact that we didn't have a proper grid layout system in CSS. Now, thankfully, we do.

Using CSS Grid, we can create grids of virtually infinite permutations and position child elements wherever we want them, regardless of their source order. What's more, Grid even makes provisions for when additional items are added and adapts itself to the needs of the content. This might sound hyperbolic, so let's not waste any more text. Let's get to it.

Basic Grid syntax

In the simplest terms, to use Grid, we need to tell the browser the following:

- How many rows and columns our grid should have
- How those rows and columns should be sized
- Where we want to place the items of our grid
- What should happen when the size of the grid changes or more items are added to the grid

Relating that to the browser is, therefore, just a matter of understanding the terminology.

Grid-specific concepts and terminology

The first concept to understand is **explicit** and **implicit** item placement. The grid layout you define in your CSS with columns and rows is the explicit grid; it's the layout of the items you have explicitly defined. Meanwhile, the implicit grid is the grid placement of items that happens when additional items you didn't foresee also get added to the grid. The placement of these new items is implied by the layout of your explicit grid. Don't worry if that doesn't make sense yet; it will!

The next concept that commonly confuses people (it certainly did me) is that grid lines are on either side of the grid items. The bit in the middle, between the lines, is referred to as a grid **track**. Where two tracks from different directions intersect is where a **grid area** is formed. This grab from the Dev Tools of Chrome should help illustrate the point:

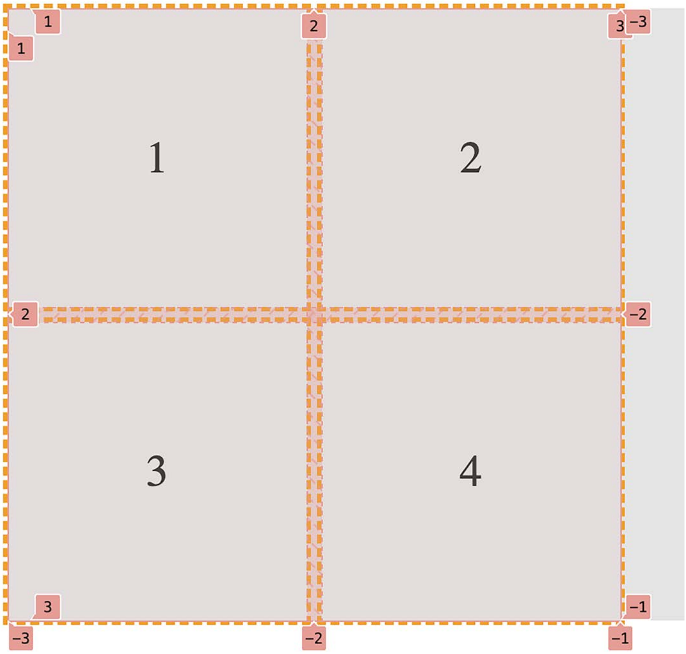


Figure 5.1: Be aware that grid lines sit on either side of tracks, where the content lives

A key takeaway is that when you place items within a grid, you can do so by referencing the grid lines (which, therefore, implies a grid area) or the grid areas themselves, if they are named.

There is no limit, theoretically, to the number of tracks a grid can have. However, browsers are at liberty to truncate grids. If you set a value beyond the browser's limit, the grid will be truncated to that limit. Practically, I can't imagine hitting a browser's limit being a likely scenario, but I'm mentioning it here for completeness.

Setting up a Grid Layout

Here is the introduction to the explicit grid section of the W3C specification (`https://www.w3.org/TR/css-grid-1/#explicit-grids`). It's worth rereading a few times as it is dense with essential information for understanding how the grid works:

> *The three properties grid-template-rows, grid-template-columns, and grid-template-areas together define the explicit grid of a grid container. The final grid may end up larger due to grid items placed outside the explicit grid; in this case implicit tracks will be created, these implicit tracks will be sized by the grid-auto-rows and grid-auto-columns properties.*
>
> *The size of the explicit grid is determined by the larger of the number of rows/columns defined by grid-template-areas and the number of rows/columns sized by grid-template-rows/grid-template-columns.*
>
> *Any rows/columns defined by grid-template-areas but not sized by grid-template-rows/grid-template-columns take their size from the grid-auto-rows/grid-auto-columns properties. If these properties don't define any explicit tracks the explicit grid still contains one grid line in each axis.*
>
> *Numeric indexes in the grid-placement properties count from the edges of the explicit grid. Positive indexes count from the start side (starting from 1 for the start-most explicit line), while negative indexes count from the end side (starting from -1 for the end-most explicit line).*
>
> *The grid and grid-template are properties that allow a shorthand to be used to set all three explicit grid properties (grid-template-rows, grid-template-columns, and grid-template-areas) at the same time. The grid shorthand also resets properties controlling the implicit grid, whereas the grid-template property leaves them unchanged.*

Now, I'm conscious that when you haven't worked at all with Grid layout, it can be very intimidating, and that quoted section from the specification might seem completely opaque right now. Hopefully, when you have read this chapter through and played with Grid Layout yourself, it will make a little more sense.

Again, don't worry if little of that previous text made sense. Let's start our Grid journey in code with a very simple example. First, the world's easiest Grid Layout: four numbered boxes. It will look like this in the browser, and you can find it in the chapter code as `example_05-01`:

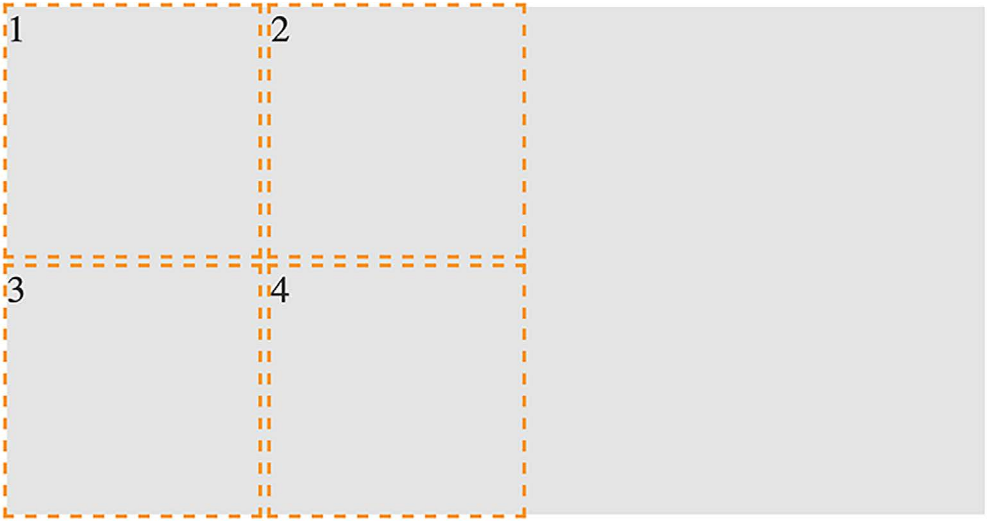


Figure 5.2: Our first grid; as simple as possible

Here is the markup:

```
<div class="my-first-grid">
  <div class="grid-item-1">1</div>
  <div class="grid-item-2">2</div>
  <div class="grid-item-3">3</div>
  <div class="grid-item-4">4</div>
</div>
```

The first thing I want you to consider is that with Grid, the markup pattern is a containing element, which is the grid, and the elements of the grid are the direct children. Write the markup for grid child elements in the order that makes the most sense for the content; Grid can place them visually wherever you need them. Here is the related CSS:

```
.my-first-grid {
  display: grid;
  grid-gap: 10px;
  grid-template-rows: 200px 200px;
  grid-template-columns: 200px 200px;
  background-color: #e4e4e4;
}
```

```
.grid-item-1 {
  grid-row: 1;
  grid-column: 1;
}
.grid-item-2 {
  grid-row: 1;
  grid-column: 2;
}
.grid-item-3 {
  grid-row: 2;
  grid-column: 1;
}
.grid-item-4 {
  grid-row: 2;
  grid-column: 2;
}
[class^='grid-item'] {
  outline: 3px dashed #f90;
  font-size: 30px;
  color: #333;
}
```

The parts to concentrate on in that CSS are the grid-specific properties. I've added some outlines and a background to make it easier to see where the grid is and the size and shape of the grid items.

We use `display: grid` to set our container as a grid and then use `grid-template-rows: 200px 200px` to set two rows, each 200px high, and `grid-template-columns: 200px 200px` to set the grid to have two 200px wide columns.

In terms of the child elements of the grid, we use `grid-row` with a number to tell Grid Layout which row to place the item in, and `grid-column` to tell it which column to place it in.

By default, the child elements of a grid remain in their standard layout type. Although our grid items belong to our grid, in this example, as they are all `div` elements, they are still computed as `display: block`. This is important to understand when we start trying to align grid items.

Let's use the alignment properties we learned about in the last chapter to try centering our grid items:

```
.my-first-grid {
  display: grid;
  grid-gap: 10px;
  grid-template-rows: 200px 200px;
  grid-template-columns: 200px 200px;
  background-color: #e4e4e4;
  align-items: center;
  justify-content: center;
}
```

When I first started with Grid Layout and tried something like this, I expected to see the numbers perfectly centered in their respective grid tracks. However, that is not the case:

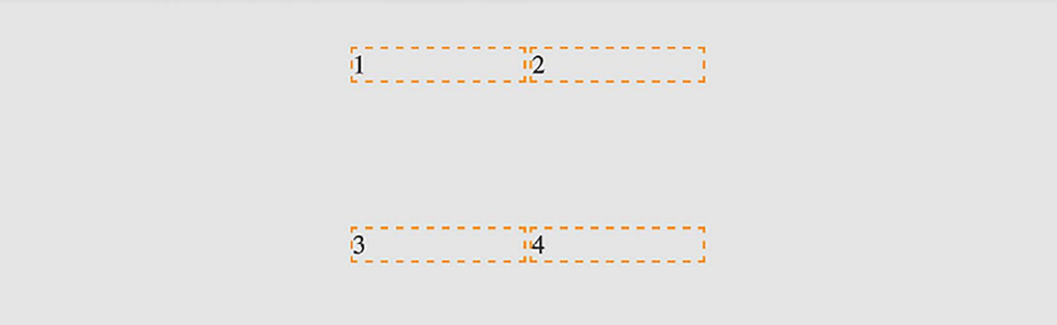


Figure 5.3: If you don't set a width, the grid will consume the available space

If we think about what we have done, it does make sense. We made a grid with two columns and two rows, each 200px, and asked for the items to be both vertically and horizontally centered. Because we have used `grid` and not `inline-grid`, the grid fills the entire width of the page, despite the fact that our grid items don't need all that space.

Let's tweak this so that the grid is only as wide as its content. Furthermore, let's center the items inside their respective grid items. To do that, we can make the items themselves either Flexbox or Grid. As this is a chapter about Grid, let's try using that and amend our code to this:

```
.my-first-grid {
  display: inline-grid;
  grid-gap: 10px;
  grid-template-rows: 200px 200px;
  grid-template-columns: 200px 200px;
  background-color: #e4e4e4;
}
```

```
[class^='grid-item'] {
  display: grid;
  align-items: center;
  justify-content: center;
  outline: 3px dashed #f90;
  font-size: 30px;
  color: #333;
}
```

If that selector with the hat symbol doesn't make sense, don't worry. We will cover that in *Chapter 6, CSS Nesting, Layers, Selectors, and More*.

We switched our container to be an `inline-grid`, set all the grid items to `display: grid`, and used the `justify-content` and `align-items` alignment properties.

That produces this result:

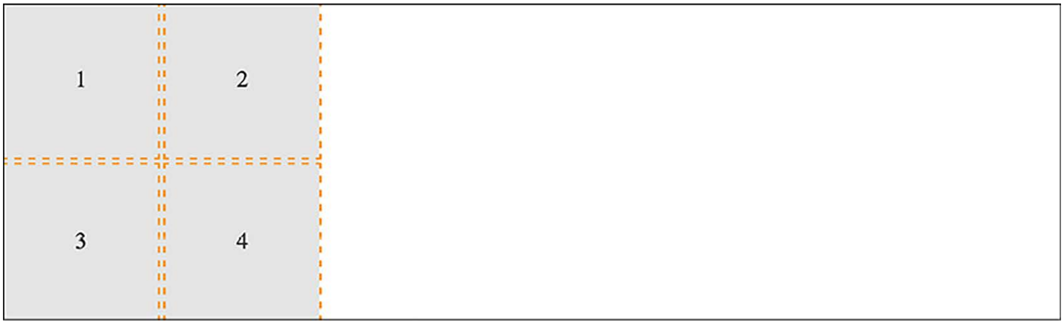


Figure 5.4: By making the child elements flex or grid, we can center their contents

You can find the code for this at `example_05-02`. As an exercise, try moving the position of the grid elements to different rows and columns by amending the `grid-row` and `grid-column` properties.

OK, a little progress has been made. Let's move on to the topic of explicit and implicit item placement.

Explicit and implicit

Earlier in the chapter, we discussed the difference between an explicit and implicit grid, an explicit grid being the structure you define in your CSS when setting up the grid. When more content is placed in that grid than you provisioned for, the implicit grid comes into effect.

Let's look at that again by extending our prior example.

We'll add in another item and see what happens:

```html
<div class="my-first-grid">
  <div class="grid-item-1">1</div>
  <div class="grid-item-2">2</div>
  <div class="grid-item-3">3</div>
  <div class="grid-item-4">4</div>
  <div class="grid-item-5">5</div>
</div>
```

With that added, this is what we get in the browser:

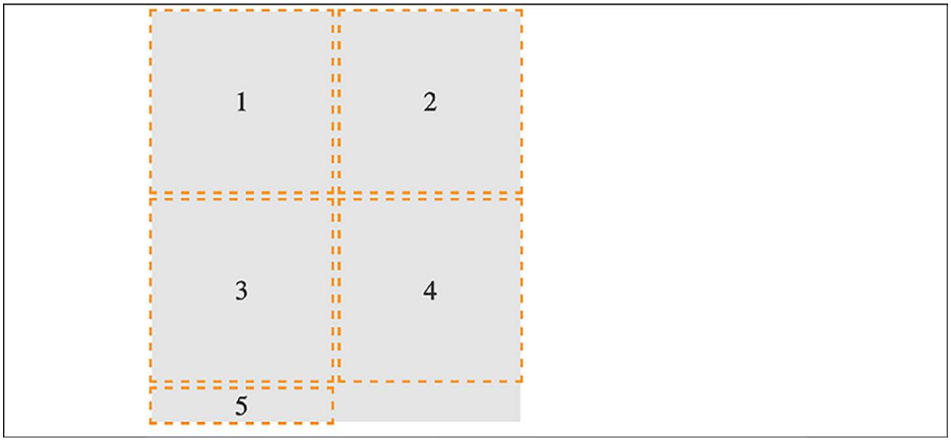


Figure 5.5: The grid has added our item, but not with the kind of proportions we were hoping for

That's sort of useful; the grid has created implicit grid lines to create an implicit track for our new item. Now, we didn't tell it what to do with that extra item, so it made the best guess. However, we can control how Grid handles items implicitly with the following properties: `grid-auto-rows` and `grid-auto-columns`.

grid-auto-rows and grid-auto-columns

Let's use `grid-auto-rows` and `grid-auto-columns` to make any extra grid items the same size as the existing ones:

```css
.my-first-grid {
  display: inline-grid;
  grid-gap: 10px;
  grid-template-rows: 200px 200px;
  grid-template-columns: 200px 200px;
```

```
    grid-auto-rows: 200px;
    grid-auto-columns: 200px;
    background-color: #e4e4e4;
  }
```

Now, without writing any extra CSS, when additional items are placed in our grid, they get the 200px × 200px sizing we have defined. Here, we have added another item in the DOM for a total of six:

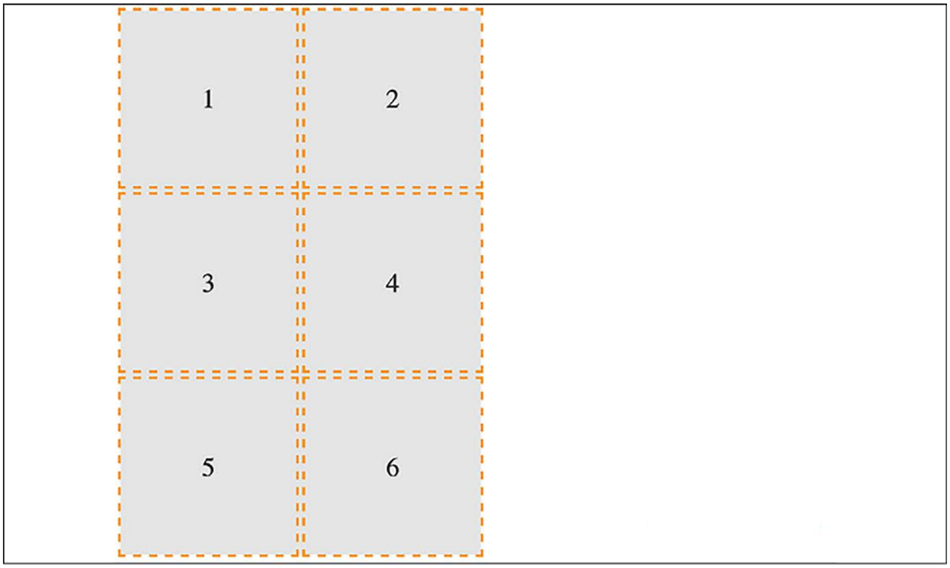


Figure 5.6: With some auto settings, extra elements can be added at a more preferable size

You can even make patterns so that the first extra item is one size and the next is another. The pattern gets repeated:

```
  .my-first-grid {
    display: inline-grid;
    grid-gap: 10px;
    grid-template-rows: 200px 200px;
    grid-template-columns: 200px 200px;
    grid-auto-rows: 100px 150px;
    grid-auto-columns: 100px 150px;
    background-color: #e4e4e4;
  }
```

Here's what it looks like:

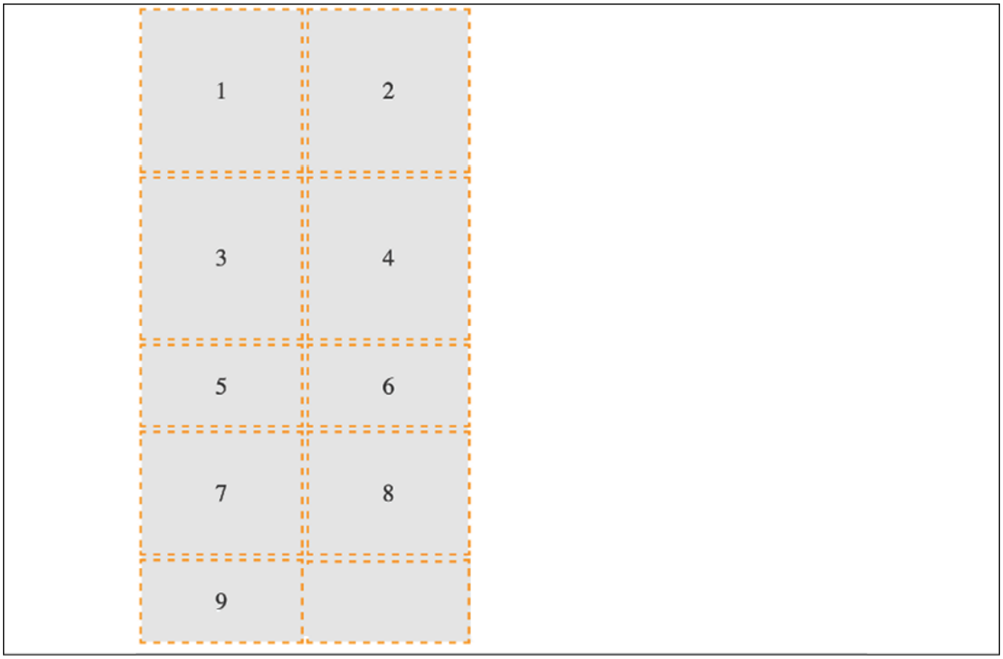


Figure 5.7: You can specify a sizing pattern for any auto-added rows or columns

Can you see how item 5 onward uses the pattern we defined in the value of the `grid-auto-rows` property? First, 5 and 6 are 100px tall, then 7 and 8 are 150px, and then back to 100px for 9.

So far, you can see that the grid items are flowing vertically down the page. You can easily switch this to flow across the page instead! You can play about with this code in `example_05-03`.

grid-auto-flow

The `grid-auto-flow` property allows you to define the direction that any implicitly added items flow inside the grid. Use a value of `column` when you want the grid to add extra columns to accommodate extra items so it flows horizontally, and use a value of `row` when you want the grid to add extra rows and flow vertically.

Let's amend our example by adding `grid-auto-flow: column` to make the items flow across the page instead of down:

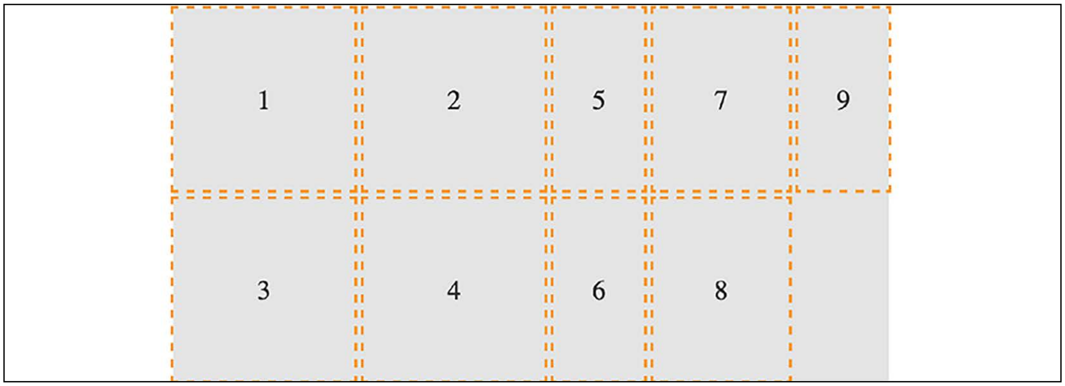


Figure 5.8: Switching the grid to add extra items horizontally

There is an additional `dense` keyword that can be added so that we have `grid-auto-flow: column dense` or `grid-auto-flow: row dense` – we'll look at that shortly.

Placing and sizing Grid Layout items

So far, each item we have added to a grid has taken up a single grid area. We are going to start a new example now (you can find it in the code as `example_05-04`). This grid will have 20 grid items. The text content of each item in the grid is merely their order in the DOM and a random food. There are quite a few new things to go over in the CSS. Before we go through each new thing step by step, take a look at the code and the screenshot and see how much of it you can make sense of before reading on.

It's also worth mentioning that I've purposely mixed up the use of white space in the values. You can write `grid-row: 6 / span 2` or `grid-row: 6/span 2` – either is just as valid. Just pick which you prefer.

Here's the markup:

```
<div class="container">
  <div class="grid-item1">1. tofu</div>
  <div class="grid-item2">2. egg plant</div>
  <div class="grid-item3">3. onion</div>
  <div class="grid-item4">4. carrots</div>
  <div class="grid-item5">5. swede</div>
  <div class="grid-item6">6. scones</div>
```

```
    <div class="grid-item7">7. cucumber</div>
    <div class="grid-item8">8. carrot</div>
    <div class="grid-item9">9. yam</div>
    <div class="grid-item10">10. sweet potato</div>
    <div class="grid-item11">11. peas</div>
    <div class="grid-item12">12. beans</div>
    <div class="grid-item13">13. lentil</div>
    <div class="grid-item14">14. tomato</div>
    <div class="grid-item15">15. butternut squash</div>
    <div class="grid-item16">16. ham</div>
    <div class="grid-item17">17. pizza</div>
    <div class="grid-item18">18. pasta</div>
    <div class="grid-item19">19. cheese</div>
    <div class="grid-item20">20. milk</div>
  </div>
```

Here is the CSS:

```
  .container {
    font-size: 28px;
    font-family: sans-serif;
    display: grid;
    gap: 30px;
    background-color: #ddd;
    grid-template-columns: repeat(4, 1fr);
    grid-auto-rows: 100px;
    grid-auto-flow: row;
  }
  [class^='grid-item'] {
    outline: 1px #f90 dashed;
    display: grid;
    background-color: goldenrod;
    align-items: center;
    justify-content: center;
  }
  .grid-item3 {
    grid-column: 2/-1;
  }
```

```
.grid-item6 {
  grid-row: 3/6;
  grid-column: 3 / 5;
}
.grid-item17 {
  grid-row: 6 / span 2;
  grid-column: 2/3;
}
.grid-item4 {
  grid-row: 4 / 7;
}
```

And here is what we get in the browser:

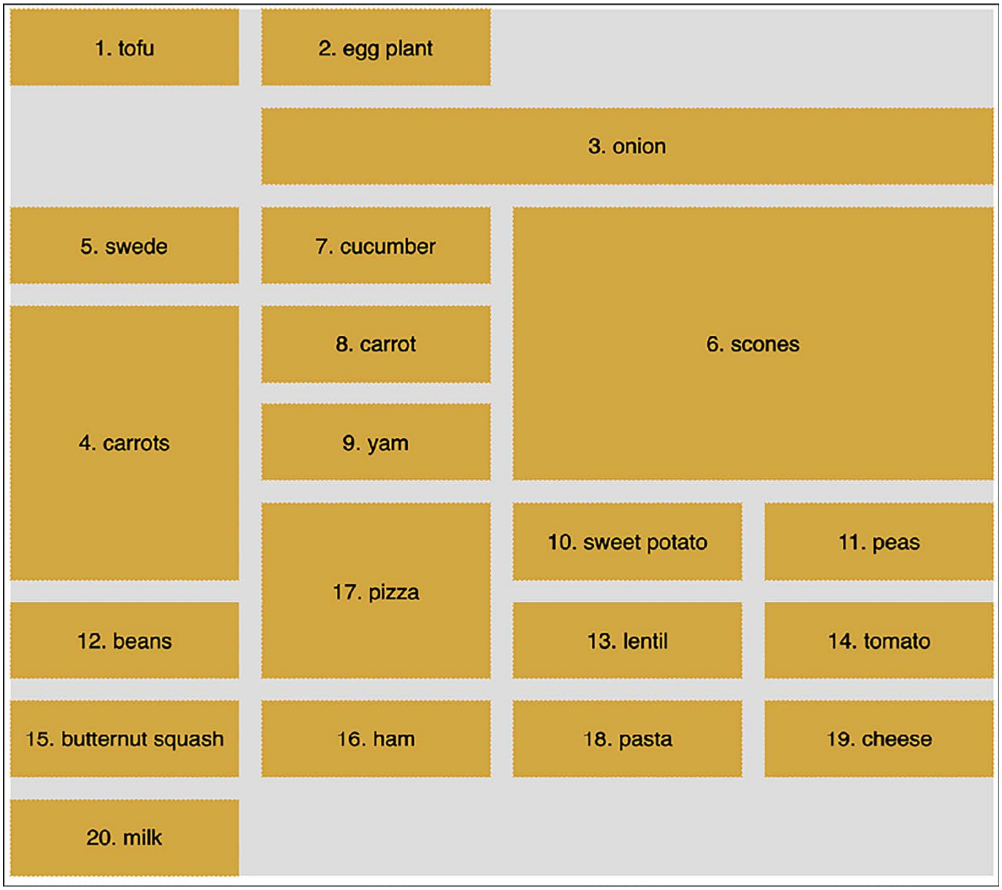

1. tofu
2. egg plant
3. onion
5. swede
7. cucumber
6. scones
8. carrot
4. carrots
9. yam
10. sweet potato
11. peas
17. pizza
12. beans
13. lentil
14. tomato
15. butternut squash
16. ham
18. pasta
19. cheese
20. milk

Figure 5.9: Grid items sized arbitrarily

We've introduced a few new things here. Let's cover each in turn, starting with how we have achieved a nice uniform gap between grid elements.

gap

I've used `grid-gap` in some of the prior code samples, but I've not explained it. In fact, the first thing I should tell you is that you can just use `gap` instead of `grid-gap`, as it works in the same way here as it does in Flexbox, which we looked at in the previous chapter. That is what I'll likely be using in the code examples going forward.

The `gap` property lets you specify a gap between grid tracks. It is actually shorthand for both `row-gap` and `column-gap`. Just like when you specify a margin with two values, the first value applies to the top and bottom (row), and the second to the left and right (columns). If you specify a single value, as we have, it applies to both.

It's especially useful because, unlike dealing with padding or margin when laying things out, you don't need to worry about undoing it again on the final item; `gap` only applies between items. And remember, you should also be aware that all modern browsers allow you to apply `gap` to Flexbox items too!

repeat

If you were making a grid with 30 identical columns, it would get a little tiring having to write `auto` 30 times, for instance, `grid-template-columns: auto auto auto auto auto auto...;` in fact, I got bored just writing that!

Thankfully, the Grid Layout specification writers have blessed us with the `repeat()` function. As you might have guessed, the `repeat()` function provides a convenient way of stamping out the need for any number of items. In our example, we have used it to create four columns, all `1fr` in width:

```
repeat(4, 1fr);
```

Like all CSS functions, we pass arguments to the function inside the parentheses. The first value we pass to the `repeat()` function is the number of times you want something repeated, and the second value is the width of each repeated item.

Don't worry, I'll explain `fr` units in a moment; for now, just know that you can create multiple columns/rows with ease. Do you want 15 columns, all 100px wide? It's as easy as `repeat(15, 100px)`.

fr units

The `fr` unit represents “flexible length” and stands for “flex fraction.” It’s used to communicate how much of any available free space we want something to gobble up, much like the `flex-grow` unit we covered for Flexbox in *Chapter 4*, *Fluid Layout and Flexbox*.

The specification doesn’t say so, but I conceptualize `fr` as standing for “free room” when I’m thinking about a layout. In our example, we have made four columns, each taking up one portion of the available free room.

Placing items in the grid

Before this example, we have been positioning each grid item in a single grid area. However, here, we have certain grid items being assigned spans of columns or rows numerically.

Let’s take the `grid-item3` example:

```
.grid-item3 {
  grid-column: 2/-1;
}
```

Here, the `grid-column` property is being set to start at the second grid line and end at the `-1` grid line. The `-1` part looks pretty odd at first, but it is part of a very smart piece of syntax.

The first number is the start point, which is separated from the endpoint with a forward slash, `/`. Positive numbers count from the start side – the left-hand side in our column example – while negative numbers start from the end side – the right, in this instance. So, `-1` basically just means the last grid line. So, this nice terse syntax just reads: “Start at line 2 and go to the end.”

There's an example of spanning across rows in there, too. Take a look at this one again:

```
.grid-item4 {
  grid-row: 4 / 7;
}
```

This one says, "Start at grid row line 4 and end at grid row line 7." If your preference is not telling the grid where something ends and more about how much space something should span, Grid Layout has just the property for you.

span

Take a look at the CSS for `.grid-item17` now:

```
.grid-item17 {
  grid-row: 6 / span 2;
  grid-column: 2/3;
}
```

Can you see how we have done something a little different with the value for `grid-row`?

Rather than stipulating a definite start and end point when placing grid items, you can give one or the other and then tell the item to span a number of rows/columns from that point either forward or backward. In our example, we told the item to start at grid line 6 and span 2 rows.

I find going backward a little more confusing, but that might just be me! However, to illustrate, we could achieve the same visual effect for our example by changing that line to `grid-row: span 2 / 8`. In this instance, the definite point we have is the endpoint, so we are telling the grid to make the item start at grid row 8 and span 2 back from there.

dense

Remember when we looked at `grid-auto-flow`, I mentioned the `dense` keyword? Well, this is the perfect opportunity to show you what that does. I'll amend the value of the `grid-auto-flow` property to this: `grid-auto-flow: row dense;`. This is what it does in the browser:

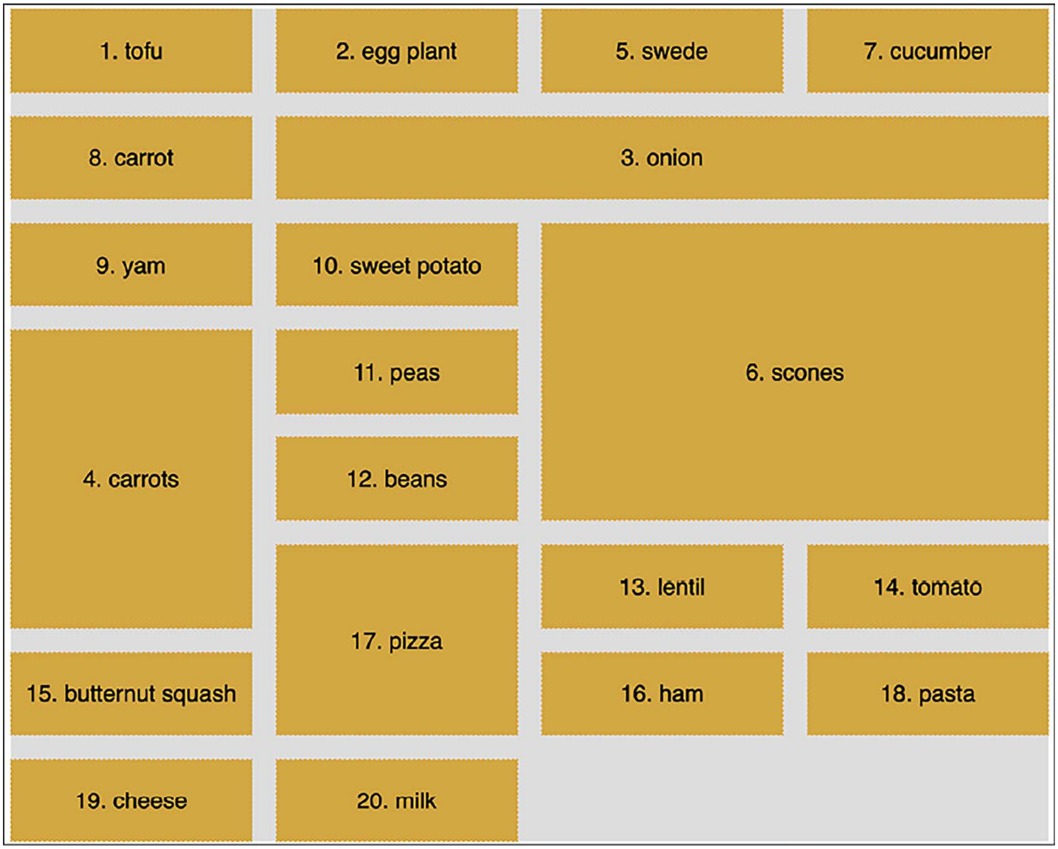

1. tofu
2. egg plant
5. swede
7. cucumber
8. carrot
3. onion
9. yam
10. sweet potato
6. scones
11. peas
4. carrots
12. beans
13. lentil
14. tomato
17. pizza
15. butternut squash
16. ham
18. pasta
19. cheese
20. milk

Figure 5.10: The dense keyword rearranges grid items so that gaps are removed

Can you see how the gaps have been filled? That's what `dense` does for you. However, while this might seem more aesthetically pleasing, there is a cost. The reason the items are numbered is that I wanted to highlight to you that using `dense` tells the Grid Layout algorithm to move things, visually, from their source order to any available space. Notice how, in this instance, the items are not necessarily in source order? That's the cost of the visual nicety.

So far, we have been laying things out using the numbers of the grid lines. However, if you would rather name the grid lines, that is something you can do too.

Named grid lines

Grid Layout allows authors to work with grids in a number of ways. For example, if you'd rather work with words than numbers, it's possible to name grid lines. Consider a three-column by three-row grid.

You can find this example in the book's code as `example_05-05`.

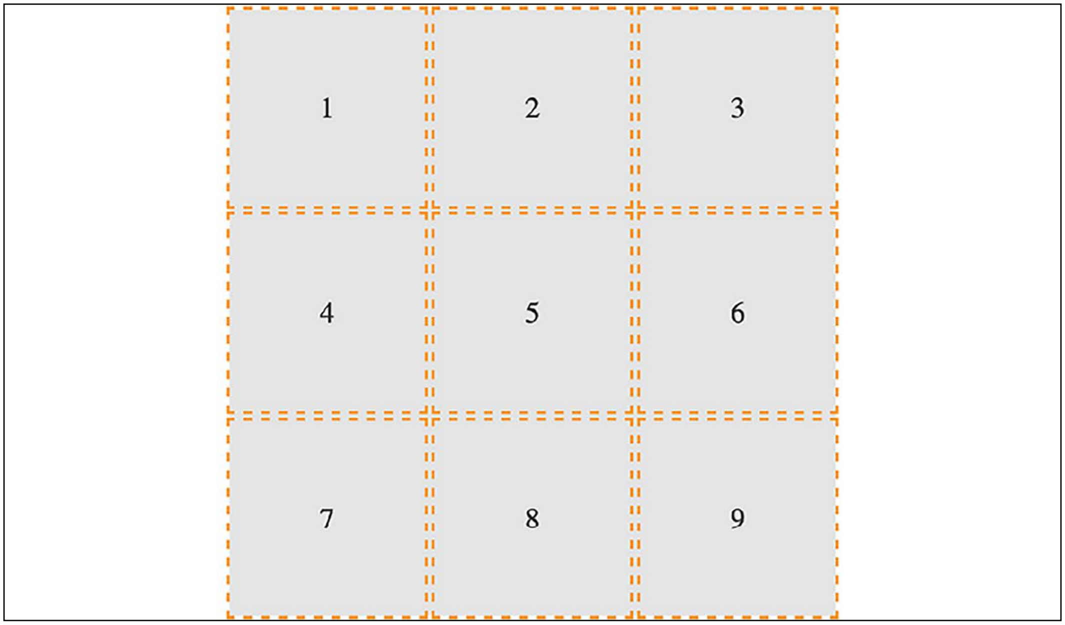


Figure 5.11: We will use named grid lines to rearrange our elements

Here's our markup:

```
<div class="my-first-grid">
  <div class="grid-item-1">1</div>
  <div class="grid-item-2">2</div>
  <div class="grid-item-3">3</div>
  <div class="grid-item-4">4</div>
  <div class="grid-item-5">5</div>
  <div class="grid-item-6">6</div>
  <div class="grid-item-7">7</div>
  <div class="grid-item-8">8</div>
  <div class="grid-item-9">9</div>
</div>
```

We set the grid up with this rule. Note the words in square brackets:

```
.my-first-grid {
  display: inline-grid;
  grid-gap: 10px;
  grid-template-columns: [left-start] 200px [left-end center-start] 200px
    [center-end right-start] 200px [right-end];
  grid-template-rows: 200px 200px 200px;
  background-color: #e4e4e4;
}
```

What we are doing inside the square brackets is giving a name to the grid line. In this instance, we have named the first column grid line `left-start`, and the one after the first column, we have named `left-end`. Notice that, in the center grid line, we have assigned two names: `left-end` and `center-start`. We can do this by space-separating the names. In this situation, it makes sense because that grid line is both the end of the left column and the beginning of the center one.

Let's amend `grid-template-row` and add some named grid lines there too:

```
grid-template-rows: [top-start] 200px [top-end middle-start] 200px
  [middle-end bottom-start] 200px [bottom-end];
```

Here's an example of how we can use these names to position grid items instead of numerical values. This is just the first three of the items we will see in the following diagram:

```
.grid-item-1 {
  grid-column: center-start / center-end;
  grid-row: middle-start / middle-end;
}
.grid-item-2 {
  grid-column: right-start / right-end;
  grid-row: bottom-start / bottom-end;
}
.grid-item-3 {
  grid-column: left-start / left-end;
  grid-row: top-start / middle-start;
}
```

In the example code, I have set each grid item to a random position using this technique. You can see how the preceding three are placed in this figure:

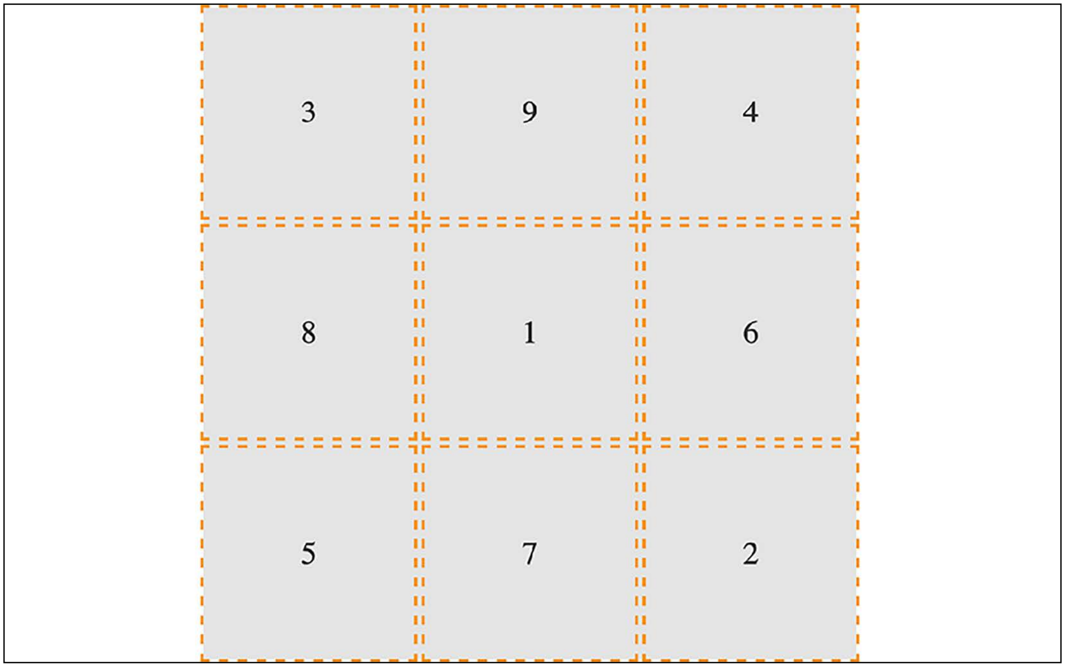


Figure 5.12: You can move things around just as easily with named grid lines

In specification terms, the name we assign to a grid line is known as a **custom ident**. Because they are just words, avoid using terminology that might interfere with grid keywords. For example, don't start naming grid lines "dense," "auto-fit," or "span"!

Grid Layout has an extra nicety you can make use of when you use named grid lines. If you append your names with `-start` and `-end`, as we have in our example, then `grid` automagically (yes, I know, that's not a real word) makes you a named grid area. Wait, what? Yep, that means that once you have named your grid lines, you can place items in your grid with a one-line `grid-area`. To prove that point, here are just the first three `grid-item` rules from earlier rewritten this way:

```
.grid-item-1 {
  grid-area: middle / center;
}
.grid-item-2 {
  grid-area: bottom / right;
}
.grid-item-3 {
  grid-area: top / left;
}
```

I've gotten a little bit ahead of myself now, as I have introduced `grid-area` without any explanation. Let's cover that now.

grid-template-areas

Yet another way you can work with Grid Layout is to create grid template areas to establish the areas of your grid. Let's rework our prior example, and we will remove the named grid lines entirely, starting again with this basic CSS for the grid.

This is `example_05-06` in the code:

```
.my-first-grid {
  display: inline-grid;
  gap: 10px;
  grid-template-columns: 200px 200px 200px;
  grid-template-rows: 200px 200px 200px;
  background-color: #e4e4e4;
}
[class^='grid-item'] {
  display: grid;
  align-items: center;
  justify-content: center;
  outline: 3px dashed #f90;
  font-size: 30px;
  color: #333;
}
```

Now, we will define our grid template areas like this, which we will add to the `.my-first-grid` rule:

```
grid-template-areas:
  'one two three'
  'four five six'
  'seven eight nine';
```

With `grid-template-areas`, we can define rows and columns very simply. A row is defined with quotes (double or single), with the names of each column space-separated inside. Each row in the grid is just another pair of quotes with custom idents inside.

You can use numbers for the start of each grid area, but you then need to **escape** them when you reference them.

Escaping a character, in computer terms, is a means of invoking an alternative interpretation of the character. It is a way of being more explicit about the meaning you would like a certain character to have that could potentially be interpreted in multiple ways.

For example, if one of our areas was named “9,” it would have to be referenced like this: `grid-area: "\39;"`. I find that too burdensome, so I suggest using a string for each area or, at least, starting each custom ident with an alpha character.

And that means we can position our items with `grid-area` like this:

```
.grid-item-1 {
  grid-area: five;
}
.grid-item-2 {
  grid-area: nine;
}
.grid-item-3 {
  grid-area: one;
}
```

Admittedly, this is a simplistic example, but hopefully, you can imagine a more useful scenario: perhaps a blog layout with a header, left-hand sidebar area, main content area, and footer. You might define the necessary `grid-template-areas` like this:

```
grid-template-areas:
  'header header header header header header'
  'side side main main main main'
  'side side footer footer footer footer';
```

As the specification (the link for the relevant section follows) states that any white space character fails to produce a token, which, for our purposes, simply means delineating one area from another, you could opt to separate columns with tab characters instead of a space if you would rather, to aid in visually lining up the columns: `https://www.w3.org/TR/css-grid-1/#valdef-grid-template-areas-string`.

When you write the grid template areas, the indentation is not important, nor are the carriage returns; you could lay each row out in a long space-separated list if you liked. As long as the names for each area are inside quotes, with a white space character between each, and there is a space between each set of quotes, all is well.

Let's move on to some even more advanced Grid Layout techniques now.

auto-fit and auto-fill

`auto-fit` and `auto-fill` are "repeat-to-fill" keywords used to describe repetition to Grid.

Thanks to the similarity of their names, I can't help but feel that the `auto-fit` and `auto-fill` keywords are almost guaranteed to confuse, just like `cover` and `contain` do for background image-sizing (we will cover them in *Chapter 9*, *Stunning Aesthetics with CSS*).

The upshot of this is that I frequently have to check which one is which. Thankfully, for both our benefits, we're going to clarify what each of these does now and why you might want to use them.

Let me start with the *why*, as that covers both keywords. What if I told you that with the help of `auto-fill` or `auto-fit`, we can create a fully responsive grid that adds/removes columns based on the available size of the viewport, with no media queries needed?

Compelling, right?

Consider a 9-column grid, where each column is at least 300px wide. In a slightly back-to-front way, I want to start this by showing you the solution:

```
grid-template-columns: repeat(auto-fit, minmax(300px, 1fr));
```

And here is what that gives you in the browser in a smaller viewport:

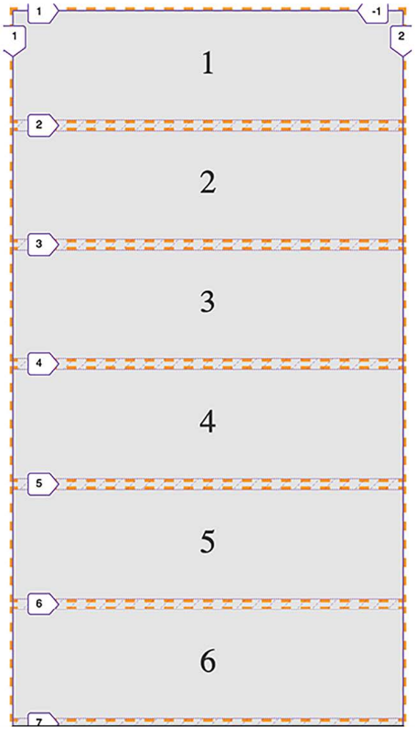


Figure 5.13: One line with Grid gives you a mobile layout…

And here is the same thing on a wider screen:

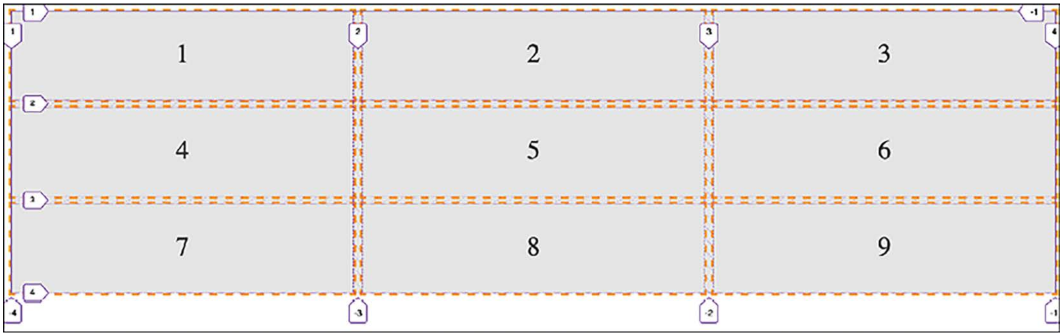


Figure 5.14: …and also a layout for wider viewports

Pretty handy, right?

Let's break down all the magic in that one-liner.

We are using `grid-template-columns`, as we have done before, to set up the columns of our grid. We are using the `repeat()` function to set up a repeating pattern of columns; however, instead of passing in a number of columns, we tell the grid to use `auto-fit`. We may have used `auto-fill` there, too, but we'll get to the differences between them in a moment. For now, we have told the browser to repeatedly create `auto-fit` columns, and we define the width of those columns using the `minmax` function.

The minmax() function

Now, if you are new to Grid, it's likely you haven't used `minmax()` before. It's a CSS function that allows you to set up a range for the browser. You specify a minimum size and a maximum size, and it computes something in between based on the available space. In our example, we are passing `minmax()` a minimum size of 300px and a maximum size of 1 `fr` (remember, it might help to think of `fr` as "free room").

> When dealing with `minmax()`, if you specify a maximum size that is smaller than the minimum, the maximum will be ignored and the minimum will be the computed value.

With that set, the grid "auto-fits" columns that are at least 300px wide and no more than the size of its content plus a `1fr` share of any remaining space.

In practical terms, this provides a responsive layout that lets our grid resize itself based on the available viewport width.

To more easily show you the difference between `auto-fit` and `auto-fill`, let's revise our one-liner to a minimum of 100px:

```
grid-template-columns: repeat(auto-fit, minmax(100px, 1fr));
```

That gives us this layout on a wide viewport:

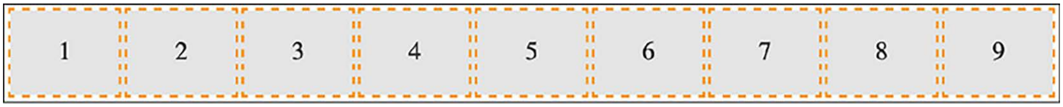


Figure 5.15: Using auto-fit will fit our content into the available space

Note how the columns span the entire width of the page. Now, we will change to `auto-fill`:

```
grid-template-columns: repeat(auto-fill, minmax(100px, 1fr));
```

Here is what that produces:

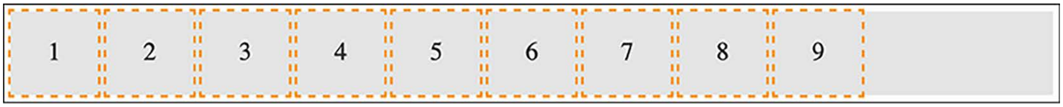


Figure 5.16: Using auto-fill, any spare space is filled up with invisible columns

Notice the space at the end? What happened there?

The key to understanding the difference comes down to whether spare columns are collapsed or not.

When the browser creates the grid in both `auto-fit` and `auto-fill`, it initially lays them out the same. However, when using `auto-fit`, any extra columns left, having laid out the content, are collapsed, leaving that space free to be distributed evenly between the items in the row. In our example (play with it in the code as `example_05-07`), because each item also has a maximum size of 1 `fr` unit, each takes up an equal portion of that free space. This results in columns that span the entire width of the space.

However, with `auto-fill`, once the items (in our example, 100px wide) are laid out, if there is any extra space, the extra empty columns are not collapsed. They remain in situ, filled up, and subsequently, the space is not free for the grid items to gobble up. The result is that we get space at the end.

There will be instances where one is more appropriate than the other; just be aware that you can achieve either.

Shorthand syntax

There are a couple of shorthand syntaxes you can use with Grid: one relatively straightforward, one less so. The first one that you'll probably find most utility for is `grid-template`.

While shorthand syntaxes can be great, my advice would be to write your grids one property at a time, at least to begin with. When you get confident enough that writing each property and value out individually becomes a chore, take the time to learn the shorthand variant.

With that advice dispensed, let's look at these two shorthand methods.

The grid-template shorthand

This allows you to set `grid-template-rows`, `grid-template-columns`, and `grid-template-areas` in one line.

So, for example, for a grid with two 200px rows and three 300px columns, we could write the following:

```
grid-template: 200px 200px / 300px 300px 300px;
```

Or, if you'd rather do it with the `repeat` function, you could write this:

```
grid-template: repeat(2, 200px) / repeat(3, 300px);
```

The part before the forward slash (/) deals with the rows and the bit after the columns. You can add `grid-template-areas` in there too if you like:

```
grid-template:
  [rows-top] 'a a a' 200px
  'b b b' 200px [rows-bottom]
  / 300px 300px 300px;
```

That computes to this in the browser:

```
grid-template-rows: [rows-top] 200px 200px [rows-bottom];
grid-template-columns: 300px 300px 300px;
grid-template-areas: 'a a a' 'b b b';
```

Personally, I find that once you start mixing in named template areas alongside numerical values, it starts to get too confusing to understand easily. Regardless, some people love the shorthand syntax (maybe you will be one of them), and you should be aware it's possible.

However, we're going to take it up another notch now and deal with the `grid` shorthand.

The grid shorthand

The other shorthand syntax, `grid`, lets you define a grid all in one line.

Using this one property, you can set the properties controlling the explicit portion of a grid (`grid-template-rows`, `grid-template-columns`, and `grid-template-areas`), as well as the properties controlling the implicit part of a grid: `grid-auto-rows`, `grid-auto-columns`, and `grid-auto-flow`.

An essential concept to hold onto when writing the `grid` shorthand is that a grid can only grow implicitly with either rows or columns, not both.

At first, that might seem odd, but if you think about it, how would a grid that could add both rows and columns implicitly lay items out? How would it decide whether to add rows or columns?

So, with that in mind, we can endeavor to wrap our heads around the `grid` shorthand.

The `grid` shorthand is not for the faint-hearted, so don't be discouraged if it spits you back out a few times before you get to grips with it. I consider myself pretty familiar with CSS (I know, you'd hope so, right?), but it took me hours rather than minutes to feel confident with what the `grid` shorthand syntax was doing.

I hope you know your Backus-Naur form, because here is how the property is described in the specification:

```
<'grid-template-rows'> / [ auto-flow && dense? ] <'grid-auto-columns'>? [
auto-flow && dense? ] <'grid-auto-rows'>? / <'grid-template-columns'>
```

Simple, right?

I kid, of course. At this point, when trying to understand a specification, I tend to search out this piece on understanding CSS specifications: `https://www.smashingmagazine.com/2016/05/understanding-the-css-property-value-syntax/`.

After rereading that, I'll now do my best to distill that piece of the specification into something more human-friendly.

What that means is, the `grid` shorthand can accept any one of three different sets of syntax as its value.

The grid shorthand value – option one

This is the same value that you would use if you were using `grid-template`. For example, here is a grid with two 100px rows and three 200px columns:

```
grid: 100px 100px / 200px 200px 200px;
```

Just like with our `grid-template` examples from earlier, you can also use `grid-template-areas` if you like.

The grid shorthand value – option two

This involves specifying a set of lengths for explicit grid rows, then, separated by a forward slash, a definition for how you want to handle implicit columns. This can be `auto-flow` to set `grid-auto-rows`, with the option of setting `grid-auto-flow` by adding `dense` too. Alternatively, you could add a length value for the width of the columns if you want to set `grid-template-columns` instead.

Phew, that's a lot to compute. Take a look at some examples.

So, for a grid with two explicit 100px rows and any number of explicit columns that are 75px wide, use the following:

```
grid: 100px 100px / repeat(auto-fill, 75px);
```

With this grid, should you have too many items, they will spill over into the implicit rows with a default size of `auto`.

In the browser, that shorthand will compute to the following:

```
grid-template-rows: 100px 100px;
grid-template-columns: repeat(auto-fill, 75px);
grid-template-areas: none;
grid-auto-flow: row;
grid-auto-rows: auto;
grid-auto-columns: auto;
```

Let's try another. Say we want to lay out a grid that has only one row, 100px high, but any number of columns, potentially running off the side of the container:

```
grid: 100px / auto-flow;
```

That rule computes to the following:

```
grid-template-rows: 100px;
grid-template-columns: none;
grid-template-areas: none;
grid-auto-flow: column;
grid-auto-rows: auto;
grid-auto-columns: auto;
```

It's also worth knowing that, when you use the `grid` shorthand, you are resetting all of the grid-related values the shorthand deals with back to their initial state. You can see that if you look at the computed values of styles in the developer tools of your browser.

The grid shorthand value — option three

The final syntax you can pass to `grid` is effectively the opposite of option two. This time, you set `grid-auto-flow` to handle implicit rows (with an optional `dense`) with an optional `grid-auto-rows` value for the size of the rows. Then, after the forward slash, we handle `grid-template-columns`.

With this kind of value, we are getting our grid to lay out in rows when needed rather than in columns, as in the previous option. Here are some examples.

How about a grid that creates as many 100px rows as needed and 5 columns that occupy `1fr` each?

```
grid: auto-flow 100px / repeat(5, 1fr);
```

That computes to this:

```
grid-auto-columns: auto;
grid-auto-flow: row;
grid-auto-rows: 100px;
grid-template-areas: none;
grid-template-columns: 1fr 1fr 1fr 1fr 1fr;
grid-template-rows: none;
```

Or, what about a grid that creates a single column with as many 100px rows as needed for the content?

```
grid: auto-flow 100px / auto;
```

That computes to this:

```
grid-auto-columns: auto;
grid-auto-flow: row;
grid-auto-rows: 100px;
grid-template-areas: none;
grid-template-columns: auto;
grid-template-rows: none;
```

You can see that the `grid` shorthand is very powerful, but arguably, not particularly intuitive. For some people, the shorthand is preferable. For some, it is maddening. There is no right or wrong – just what is right or wrong for you.

A challenge

Now seems a decent time to set you a little challenge. Imagine if you will, we are building a leaderboard for a cake-eating competition. Each competitor is represented like this:

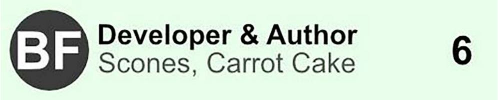


Figure 5.17 – The layout we want to build

Your task is to build this piece of UI with one wrapping element and four child elements, with every child element positioned as part of a grid. The avatar area on the left and the leaderboard position number on the right should both be fixed widths, while the center sections, with flowing text, should be flexible so that when there is more space, this area grows.

There are a number of ways to do this with Grid. Don't worry too much about font sizes and colors. Instead, concentrate on the general layout and alignment of the items, the gaps between them, and the rules that define which part can grow and which can not. I have added my take on this challenge as `example_05-12`, but that doesn't mean my way is the right way; it's merely the way I would approach it. Grid provides a lot of flexibility, so do what makes sense for you.

Allowing nested elements to take part in the grid

Once you start working with Grid, it won't be long until you hit a seemingly ridiculous shortcoming. If you are dealing with nested elements and wondering why on earth you are not able to make them align with your grid, welcome to the long list of other developers who have thought that very thing. So, what can we do about it?

It is important to be aware that when you define a grid, it is only the direct descendant elements of that grid element that can participate in the layout. The children of those direct descendants cannot.

One way we can work around this particular problem is with `display: contents`.

What the `display: contents` declaration does is effectively make the layout aspect of the element it is set on – and crucially, not the accessibility side – disappear, allowing the children of that element to behave visually, as if the parent element didn't exist.

Let's consider this markup; try this out for yourself as we go by playing with `example_05-08`:

```html
<div class="container">
    <div class="child">Child-1</div>
    <div class="child">Child-2</div>
    <div class="parent">
        <div class="grandchild">Grandchild 3</div>
        <div class="grandchild">Grandchild 4</div>
    </div>
</div>
```

And then use this CSS:

```
.container {
    display: grid;
    grid: repeat(2, 200px) / repeat(2, 200px);
}

[class*='child'] {
    border: 2px dashed orange;
}
```

By default, that gives us something like this in the browser:

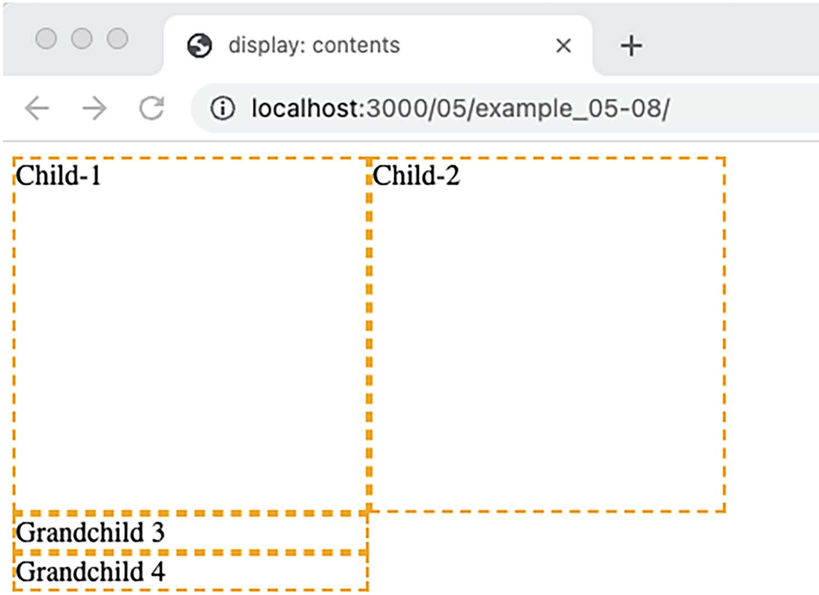
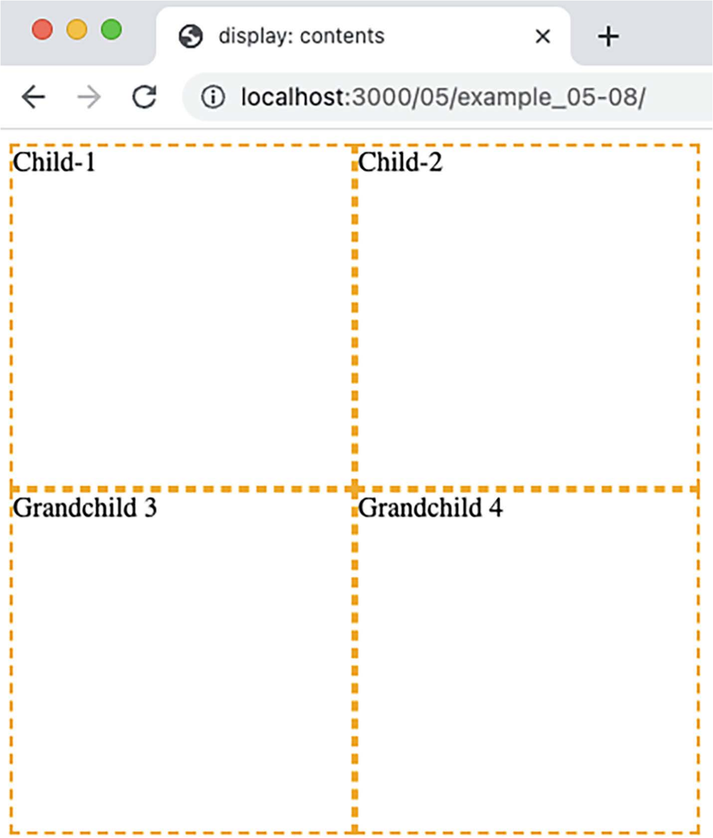

display: contents
localhost:3000/05/example_05-08/
Child-1
Child-2
Grandchild 3
Grandchild 4

Figure 5.18: By default, descendants of grid items cannot take part in the grid

You can see that the two grandchildren elements are not taking part in the grid. We can solve that like this:

```
.parent {
    display: contents;
}
```

By setting the parent element to have `display: contents`, we are allowing the element to display its contents and, therefore, allowing those elements to participate in the grid layout.



Child-1

Child-2

Grandchild 3

Grandchild 4

Figure 5.19: Setting display: contents on an element stops it from producing its own box, allowing child elements to take part in the layout

You are then able to position those elements in the grid as if the parent element didn't exist.

So, that is one way of solving the problem of grandchildren elements taking part in a grid layout. But there is another, arguably more appropriate method, and that is subgrid.

The subgrid value

Part of the Level 2 CSS Grid Layout specification deals with the `subgrid` value. As you read these words, `subgrid` will already be available to use in modern browsers. However, support may be lacking if you need to support much older browsers.

As ever, you can check support levels at `https://caniuse.com/css-subgrid`, and read the specification here: `https://www.w3.org/TR/css-grid-2/#subgrids`.

In essence, `subgrid` provides a mechanism for non-direct child elements of a grid ancestor to inherit their grid tracks. Using `subgrid` is as simple as setting `subgrid` as the value of either `grid-template-columns` or `grid-template-rows`, or both.

Let us start with the `subgrid` value by solving the same problem we solved with `display: contents` in the last section.

When using the `subgrid` value, instead, we would set the CSS for `.parent` like this (you can view the full code in `example_05-09`):

```
.parent {
    display: grid;
    grid-template-columns: subgrid;
    grid-column: span 2;
}
```

And just like that, this particular problem is solved.

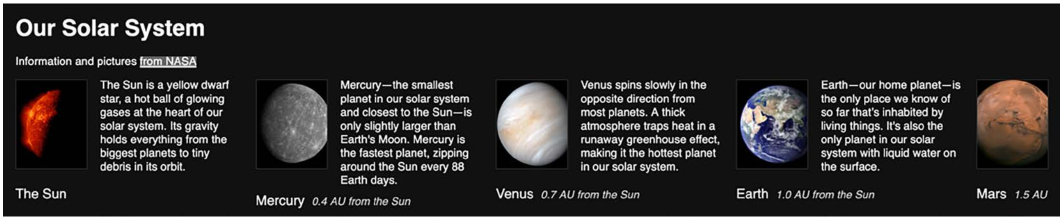
However, for solving that particular problem, that's actually more code than the prior solution with `display: contents`, but bear with me. We can understand how it works with this simple example before we look at something more involved.

First, we are making the `.parent` element a grid, and then crucially telling it that we want the template for the columns to be a `subgrid`, so that it will inherit the columns from the ancestor grid it is part of.

Next, we need to tell the browser how big this element is going to be in the ancestor grid it is part of. Here we have set it to span two columns, making it the full width of the ancestor grid, as, remember, the ancestor grid is made up of two columns.

With that in place, the `.grandchild` elements happily position themselves in the next available column. We don't need to do anything extra with them.

Let's look at one further, slightly more involved example. Consider this screenshot:



Our Solar System

Information and pictures from NASA

The Sun is a yellow dwarf star, a hot ball of glowing gases at the heart of our solar system. Its gravity holds everything from the biggest planets to tiny debris in its orbit.

The Sun

Mercury—the smallest planet in our solar system and closest to the Sun—is only slightly larger than Earth's Moon. Mercury is the fastest planet, zipping around the Sun every 88 Earth days.

Mercury *0.4 AU from the Sun*

Venus spins slowly in the opposite direction from most planets. A thick atmosphere traps heat in a runaway greenhouse effect, making it the hottest planet in our solar system.

Venus *0.7 AU from the Sun*

Earth—our home planet—is the only place we know of so far that's inhabited by living things. It's also the only planet in our solar system with liquid water on the surface.

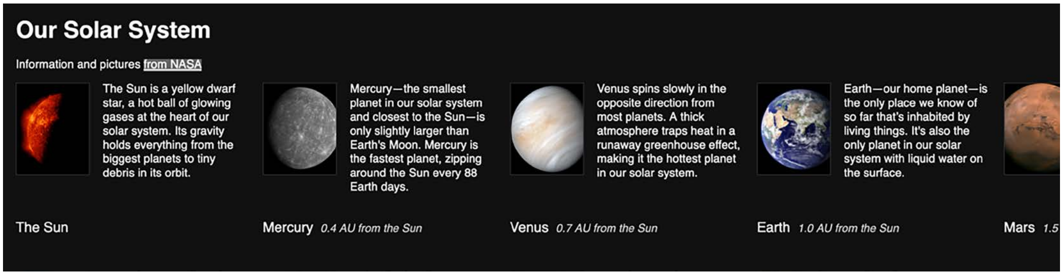
Earth *1.0 AU from the Sun*

Mars *1.5 AU*

Figure 5.20: Without subgrid, the labels at the bottom are out of alignment

Here, we have a horizontally scrolling component containing images of different planets of our solar system. Crucially, notice how, due to the information text that accompanies each planet, the caption for the image doesn't sit at the same level? While that is a problem we could fix with other means and approaches, subgrid is a perfect fit.

Here it is with our subgrid code in place. Notice how the captions below each planet line up perfectly? Let's look at how this is done.



Our Solar System

Information and pictures from NASA

The Sun is a yellow dwarf star, a hot ball of glowing gases at the heart of our solar system. Its gravity holds everything from the biggest planets to tiny debris in its orbit.

Mercury—the smallest planet in our solar system and closest to the Sun—is only slightly larger than Earth's Moon. Mercury is the fastest planet, zipping around the Sun every 88 Earth days.

Venus spins slowly in the opposite direction from most planets. A thick atmosphere traps heat in a runaway greenhouse effect, making it the hottest planet in our solar system.

Earth—our home planet—is the only place we know of so far that's inhabited by living things. It's also the only planet in our solar system with liquid water on the surface.

The Sun

Mercury *0.4 AU from the Sun*

Venus *0.7 AU from the Sun*

Earth *1.0 AU from the Sun*

Mars *1.5*

Figure 5.21: subgrid allows us to inherit the row and column tracks of a parent grid

Firstly, consider the markup. We have a containing element with the class of `.Scroll_Wrapper` to create the scrolling section, and then a number of `figure` elements, each with a class of `.Item`, that contain a planet image and the associated information.

For brevity, I'll just show you the HTML for the opening of the containing element, and one of the planets (the full code is at `example_05-10`):

```
<div class="Scroll_Wrapper">
    <figure class="Item">
        <img src="Sun.jpg" alt="Our Sun in space" />
        <figcaption class="Caption">The Sun</figcaption>
        <p>
            The Sun is a yellow dwarf star, a hot ball of glowing gases
            at the heart of our solar system. Its gravity holds everything
            from the biggest planets to tiny debris in its orbit.
        </p>
    </figure>
```

To set up the outer grid on the container, we have this:

```
.Scroll_Wrapper {
    display: grid;
    width: 100%;
    overflow-y: hidden;
    overflow-x: auto;
    gap: 0 20px;
    grid-template-rows: auto auto;
    grid-auto-columns: auto;
    grid-auto-flow: column;
}
```

Those are all grid-related properties you have already seen in this chapter. Now, as we are interested in positioning the contents of each planet, our `.Item` element, against that grid, we need to introduce `subgrid`.

We need our element to be a grid itself and, crucially, we need the rows to inherit from the nearest ancestor grid:

```
.Item {
    grid-row: 1/-1;
    display: grid;
    grid-template-rows: subgrid;
    grid-template-columns: 100px 200px;
    gap: 0 20px;
    margin: 0;
}
```

As `.Item` is itself part of an outer grid, we need to tell it how it should be positioned in that grid, so setting `grid-row: 1/-1` tells it to start at the first row and end at the last.

We then set the rows for this item's own grid with `grid-template-rows`, but use `subgrid` to get the row layout from the parent (`.Scroll_Wrapper`).

With that in place, we can set the row and column positions of child elements inside the grid as we would normally. But the "magic" part is that because our rows are connected to the outer grid via the `subgrid` value, it means they are sized with the largest child in mind, which, in turn, means that all the internal elements of each `.Item` can line up.

When you hit these kind of child element problems with Grid, if you simply need to remove a containing element as if it wasn't there, opt for `display:contents`. If, however, you need to keep the containing element – perhaps it has a background color or image – then `subgrid` is the best choice.

This is but one simple use case of a layout conundrum that can be easily solved with `subgrid`.

Summary

If you are fairly new to building on the web, you almost have an advantage when it comes to learning CSS Grid Layout: a "beginner's mind," if you like. For people who have been laying things out with other techniques over the years, the difficulty, in some ways, is undoing what you already know about layout with CSS.

Having read this chapter, you should have some understanding of what is possible with Grid Layout and how you might employ it.

Plus, with `subgrid` more recently joining the existing capabilities of CSS Grid, we now have an incredibly powerful, complete, and expressive method of laying out elements on our page with CSS.

I also ask that, if you were in any way successful with the exercises in this chapter, you congratulate yourself; there's a lot to consider when you first start using Grid. If you accomplished something with your first effort, you did admirably.

At this point, I'll also reiterate that Grid is tricky at first. It comes with a lot of possibilities but also quite a few new pieces of terminology and concepts. Expect to take a few runs at it the first time you use it.

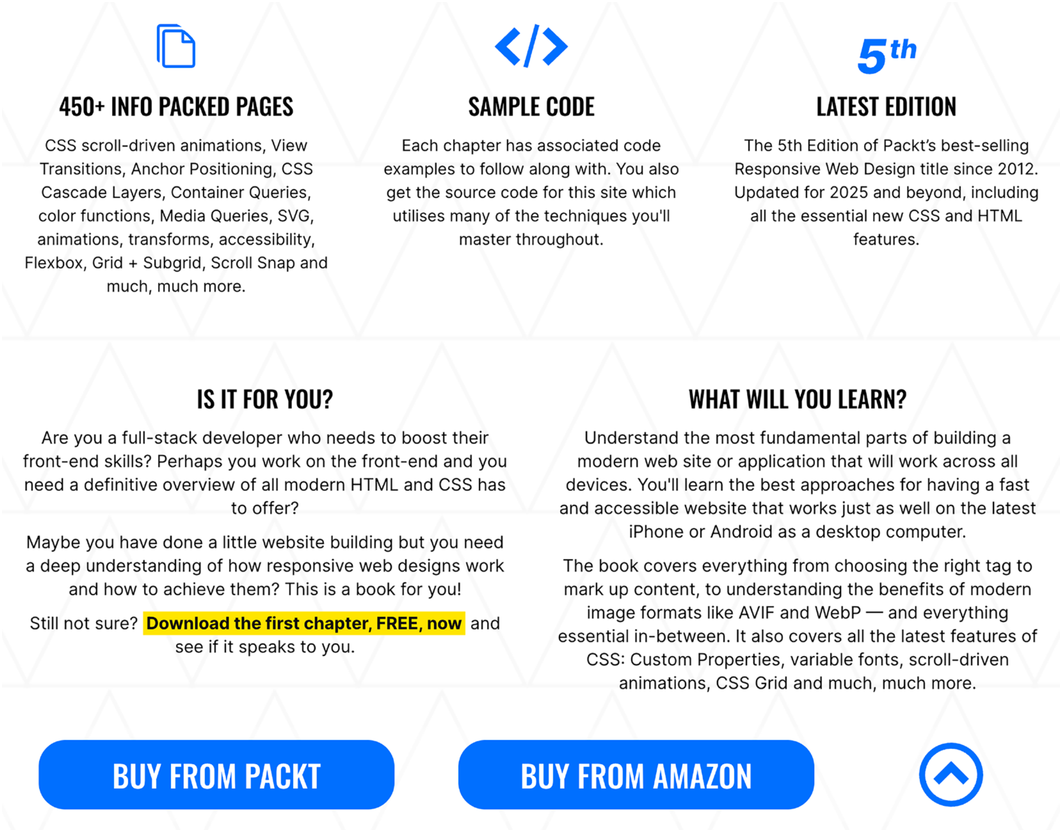
Personally, I found the easiest approach was to start remaking a few things I already had built in Flexbox with Grid. Start simple, and write the style of syntax that makes the most sense for you. I still rarely write the shorthand versions as, despite the attractiveness of a terse one-liner, I find them hard to interpret and understand at a glance. For me, clarity trumps brevity.

Regardless of the approach you take, I promise that, once you get competent, CSS Grid will pay you back handsomely for your investment.

These last two chapters, on Flexbox and Grid, have covered single topics in some detail. The next chapter is going to be more broad: a veritable grab-bag of CSS goodies. Turn the page, and let's move on to *Chapter 6*, *CSS Nesting, Layers, Selectors, and More*.

Applying what you have learned so far

As an exercise, consider this section of the `https://rwd.education` website:



450+ INFO PACKED PAGES

CSS scroll-driven animations, View Transitions, Anchor Positioning, CSS Cascade Layers, Container Queries, color functions, Media Queries, SVG, animations, transforms, accessibility, Flexbox, Grid + Subgrid, Scroll Snap and much, much more.

SAMPLE CODE

Each chapter has associated code examples to follow along with. You also get the source code for this site which utilises many of the techniques you'll master throughout.

LATEST EDITION

The 5th Edition of Packt's best-selling Responsive Web Design title since 2012. Updated for 2025 and beyond, including all the essential new CSS and HTML features.

IS IT FOR YOU?

Are you a full-stack developer who needs to boost their front-end skills? Perhaps you work on the front-end and you need a definitive overview of all modern HTML and CSS has to offer?

Maybe you have done a little website building but you need a deep understanding of how responsive web designs work and how to achieve them? This is a book for you!

Still not sure? **Download the first chapter, FREE, now** and see if it speaks to you.

WHAT WILL YOU LEARN?

Understand the most fundamental parts of building a modern web site or application that will work across all devices. You'll learn the best approaches for having a fast and accessible website that works just as well on the latest iPhone or Android as a desktop computer.

The book covers everything from choosing the right tag to mark up content, to understanding the benefits of modern image formats like AVIF and WebP — and everything essential in-between. It also covers all the latest features of CSS: Custom Properties, variable fonts, scroll-driven animations, CSS Grid and much, much more.

Figure 5.22: Can you use your grid knowledge to lay out this section?

If you look in the `Start` folder for this chapter's code, you will see that, currently, those sections are just one under the other. Try to amend the code for that section using Grid Layout. Perhaps think about how you could easily have it showing one or two of those sections when screen space is limited, and the layout that is shown in the preceding screenshot when space allows.

And what about if the headings for those sections wrap? Do you want to make use of the `subgrid` value to ensure they stay visually aligned?

Part 2

Core Skills for Effective Frontend Web Development

If you've read from the beginning to this point, you should have a solid understanding of the fundamentals needed for responsive web design. In fact, you're likely a step ahead in that you will be building your designs using the most recent layout mechanisms, Flexbox and Grid.

If we just needed to cover responsive web design, at this point, my work here would be pretty much done. But there is so much more we can look at.

This next part of the book aims to take you into more advanced and specialty areas. It's not every day you will need to reach for a `clamp` function, or know when and why you would employ a `where()` selector. However, in this part, we will be going into greater depth, understanding what, primarily, CSS can offer you to solve the everyday challenges that are going to come your way as a developer.

Again, don't think you need to remember all the details of this stuff. Take it in, get a grasp of what's possible, and know that you can dip back in the future, when you remember that "I think there's a way to do that with CSS." There's every chance that there probably is!

6

CSS Nesting, Layers, Selectors, and More

In the last few years, CSS has enjoyed a raft of new features. Some enable us to animate and transform elements. Others allow us to manipulate color, gradients, and produce mask and filter effects, and others still allow us to bring SVG elements to life.

We will get to all those capabilities and more in the next few chapters. Firstly, I think it will be useful to look at more recent improvements to some of the fundamentals of CSS.

No one can know every nuance, capability, and syntax in the CSS language. I've been working with CSS for two decades and, on a weekly basis, I still discover something new – or, just as likely, rediscover something I'd forgotten! As such, I don't feel that trying to know every possible CSS property and value permutation is actually a worthy pursuit. Instead, I think it's more sensible to develop a good grasp of what's possible and what capabilities exist that solve the most common problems, and then delve into the details of those capabilities if and when you need them.

As such, we are going to concentrate in this chapter on some of the broader new capabilities of CSS – on how you can write and organize the language, and the fundamentals of the units and selectors I find most practically useful when building responsive web designs. I'm hoping you'll then have the requisite knowledge and appreciation of what's available to solve most problems that come *your* way when developing a responsive web design.

As there are quite a few topics to cover, they have been grouped as follows:

CSS nesting:

- CSS nesting and its origins
- Examples of syntax and its use cases
- The `&` parent selector
- Nesting media queries

CSS cascade layers:

- What cascade layers are and the problems they solve
- The syntax and typical applications of cascade layers
- The rules for layer ordering
- Adding imported style sheets to a layer
- Understanding the importance of `!important`

Selectors, units, and capabilities:

- `::before` and `::after` pseudo-elements
- Attribute selectors and substring matching
- Structural pseudo-classes, including `:last-child`, `:nth-child`, `:empty`, and `:not`
- Combinator selectors, including child, next sibling, and subsequent sibling
- Functional pseudo-class selectors (`:is` and `:where`)
- `@scope` selection

Viewport-related units and feature queries:

- Viewport-related length units – `vh`, `vw`, `vmax`, and `vmin` – and their dynamic variants
- Writing feature queries using `@supports` to fork CSS

As you can see, we have a lot to get through. Let's begin.

CSS nesting

If you have done any work with CSS in the last 10–15 years, you will have come across CSS pre-processors, the most enduring of which is Sass.

When Sass first appeared, the CSS language had no variables, color functions, or nesting. I don't think it's even debatable that had it not been for the popularity and ultimate ubiquity of Sass and other pre-processors such as LESS, we wouldn't now have CSS custom properties, color manipulation functions, and the topic of this section: native CSS nesting.

CSS nesting does not provide anything new in the browser. It's purely a developer experience improvement – syntactic improvements, or sugar, if you will. The native implementation is also subtly different than how Sass implements nesting; just different enough that it might catch you out at first.

Let's consider how CSS nesting works, and you can decide whether it's something you feel will improve your development experience or not.

> I was so enamored with Sass that, back in 2013, I wrote a book on it, *Sass and Compass for Designers*. Despite being over a decade old, it's still largely relevant today. You can find out more about it here: `https://sassandcompass.com`.

If you have never used Sass before, and the concept of nesting is entirely lost on you, consider CSS like this:

```
.parent {
    font-size: 16px;
}

.parent .child {
    font-size: 1em;
}

.ancestor .parent {
    font-size: 1.5rem;
}
```

In those three rules, the common selector is the `.parent` class, which is repeated in each. With nesting, we can express the relationships between selectors with all the parent, child, and sibling relationships of a selector nested between a single set of outer braces:

```
.parent {
    font-size: 16px;
    & .child {
        font-size: 1em;
    }
    .ancestor & {
            font-size: 1.5rem;
        }
    }
```

Let's unpack what's going on here. Firstly, our outer selector is `.parent`. All nested selectors have a relationship to this outer selector. The general idea of nesting is that a rule is nested inside the braces of another to express an ancestor and descendant relationship. But what is that `&` symbol for?

> All the CSS nesting examples in this chapter are viewable in the *Chapter 6* code in `example_06-01`. Where equivalent, non-nested versions exist as a comparison, they will be commented out.

Understanding the & symbol in nesting

The `&` symbol is used in nested CSS rules to represent a reference to the outermost selector. For the purposes of understanding nesting, although not an official term, I will call this the **context** selector.

To make something a child of the context selector, you can use the `&` as the first non-whitespace character of the nested selector. However, when it comes to expressing parent and child relationships between selectors, you don't have to use the `&` symbol. That prior example could be written with the `&` omitted from the first nested rule, like this:

```
    .parent {
        font-size: 16px;
        .child {
            font-size: 1em;
        }
        .ancestor & {
            font-size: 1.5rem;
        }
    }
```

And that would be interpreted by the browser in exactly the same way.

> Although something I have never found myself having to worry about, it might be worth knowing that the & selector has the same specificity as the `:is()` selector, which we will look at shortly. In short, this means that when you use the &, it will have the same specificity as the selector in the rule that has the highest specificity.
>
> You should also be aware that if you use the parent selector, &, outside of a rule, it represents the scoping root, which, for most practical purposes, is going to be your document.

Writing an ancestor relationship

By placing the & after a nested selector, we are making that nested selector be "printed" first, and then the context selector represented by the &.

The key to understanding nesting is that the & symbol is an alias for the context of the rule. Whenever you see the &, it will be replaced by the context selector in the eyes of the browser.

To drive nesting and its possibilities home, let's consider some more examples.

Adjacent sibling and data-attribute example

Here is an adjacent sibling relationship and a selector where a data attribute and class are combined. First, the non-nested version:

```
.item {
    color: goldenrod;
}

.item + .item {
    color: red;
}

[data-scones].item {
    color: rebeccapurple;
}
```

That could be written nested like this:

```
.item {
    color: goldenrod;
    & + .item {
        color: red;
    }
    &[data-scones] {
        color: rebeccapurple;
    }
}
```

In fact, we could go one step further with the nested selector when it comes to using the ampersand and do this:

```
.item {
    color: goldenrod;
    & + & {
        color: red;
    }
}
```

This would yield the same `.item + .item` selector.

Lists of nested selectors example

When you have a list of selectors nested, you can mix and match between adding an & at the beginning or not. Both `child-1` and `child-2` items are siblings here, and both get the red color applied.

```
.lists {
    color: #333;
    & .child-1,
    .child-2 {
        color: red;
    }
}
```

Nesting conditionals such as media queries

You can also nest media queries inside a selector. We mentioned this technique briefly in *Chapter 3*, *Media and Container Queries*, as it is a great way to encapsulate all the variations that happen to a selector.

Consider this:

```
.item {
    width: 100%;
    @media (min-width: 40em) {
        width: 90%;
        margin-inline: auto;
    }
}
```

Notice that with the conditional/media query, we don't need to use the & symbol; we can just directly nest it.

Whether an element needs to have styles changed due to changes above it in the DOM tree, different media situations, or we need to express new styles in a situation where it has certain sibling selectors present, with CSS nesting, we can contain it all within a single nesting rule. Here is an example:

```
.various {
    color: #333;
    @media (prefers-color-scheme: dark) {
        color: #e8e8e8;
    }
    .container-big & {
        width: 100%;
    }
    & + & {
        margin-left: 10px;
    }
}
```

Here, we have created one rule to, err..., rule them all. Every eventuality of `.various` is inside a set of curly braces. This syntax of encapsulation is very helpful in a code base. You only need to find the key selector in question, and you can then amend anything and everything that ever happens to it within that one section of code.

That's about all you need to know to start using nesting in CSS. I'm a big fan of nesting and encapsulating rules. It's a practice I got into using Sass, and I'm pleased to see it is another feature that the CSS language has adopted as its own.

Cascade layers

Cascade layers exist as a high-level tool to manage the grouping and priority of style rules. When writing CSS, I make a point of only using classes as the styling hooks to make my changes. To exemplify, if I need to style a `button`, I won't use an isolated element selector as a styling hook like this:

```
button {
    /* styles */
}
```

This is because, when using element selectors, it isn't possible to create specificity equality across all the selectors. As soon as I want a different style button elsewhere, I will need to create a more specific selector to override the styles I have already created. Instead, I would add a class to the button and do this:

```
.btn {
    /* styles */
}
```

If the nuance of why this is important when authoring projects is lost on you, I'd recommend reading my shorter book, *Enduring CSS*, free online at `https://ecss.benfrain.com` (or buy yourself an eBook/hardcopy).

However, there may certainly be scenarios where things aren't entirely within your control. To give you a common scenario where this situation plays out, consider the following.

Imagine that, despite this author's advice against using frameworks in the next chapter, you decide you want to include a CSS framework called "Shoe Lace" in your project. This framework has rules set for the display of common elements such as buttons, lists, and the like.

You include the framework CSS and get to work on your project. However, when looking at the buttons on the page, you decide you'd like to tweak the styling a little. Maybe that border needs to be a little thinner?

You look at your markup:

```
<button class="btn">Buy Scone</button>
```

You do the sensible thing and apply the lightest possible selector in your own style sheet to make a change:

```
.btn {
    border: 1px dotted goldenrod;
}
```

Then you check the change in the browser and… nothing has happened. What is going on?

Eventually, you look inside the "Shoe Lace" framework files and uncover the problem. They have a selector like this:

```
/* CSS framework code */
button.btn {
    font-size: 1.3em;
    border: 2px solid #333;
}
```

This means that every time you want to override this set of styles, you need to use a more specific selector. That's far from ideal, as now your own styles and selectors have become more specific, which is sure to lead to more issues in the future.

Cascade layers exist to solve these kinds of problems. They can also be a useful tool in organizing your own groups of styles. But before we consider that, let's take our first look at cascade layers by seeing how we could solve our prior issue:

```
@layer framework {
    /* CSS framework code */
    button.btn {
        font-size: 1.3em;
        border: 2px solid #333;
    }
}

.btn {
    border: 1px dotted goldenrod;
}
```

Just like that, our problem has been solved. Using the `@layer` at-rule establishes a cascade layer, meaning that the styles inside the layer are immediately deemed subordinate to those outside the layer. We will get into the nitty-gritty of how layers do that momentarily.

The syntax should be relatively familiar to you. In much the same way that styles inside a media query are encapsulated by a condition, `@layer` encompasses the styles within a layer. Cascade layers can also be applied to file imports, allowing you to import a style sheet into a layer. This is something quite likely to be useful if we are dealing with a third-party library. Importing a style sheet into a cascade layer is done like this:

```
@layer framework;
@import url("shoelace-framework.css") layer(framework);
```

In that instance, we still make our layer first, and then import our style sheet into it with the `layer()` function.

Right. At this point, it's time we go through some ground rules when working with cascade layers.

Naming cascade layers

Naming a cascade layer is just like naming a set of keyframes. Within reason, name your layers however it suits your project. If you like grouping code by area, you can do that. If naming by types of styles, such as "reset," "utilities," or "colors" is more your thing, then that is fine too. There is no right or wrong; it's a case of doing whatever best suits the way you work.

You can also have anonymous layers, and they will work just the same. I can't really recommend doing that, though. For how little it takes to name a layer, it will make it far easier to work with and think about if it is named.

Where to declare layers

Declare your layers near the start of your style sheet. As you can't put layers between `@import` statements, I ensure that layers get defined at the top, then any `@imports`, and then the styles. As we will see in a moment, that doesn't mean all the styles for a layer need to be nested there too. It's just for the initial setup.

When you declare layers, they work top to bottom, with the last one declared being a higher priority than the first. So, suppose we had layers declared like this:

```
@layer reset,
framework,
layout,
components;
```

The `reset` would get the lowest priority, followed by `framework`, then `layout`, with the `components` layer having the highest priority. Whitespace isn't important, so you can write them vertically or on one line.

This would work equally well:

```
@layer reset, framework, layout, components;
```

Personally, I think it's easier to reason about them listed vertically, as it enforces the order of priority, but you do you!

It is also important to remember that the first instance of a layer defines its priority for perpetuity. Anywhere you subsequently add to, or re-declare that layer, it will remain in its current priority order.

Adding rules to layers

Because you don't have to add styles to a layer when you define it, you can add rules to a layer any number of times from any location. After a few hundred lines of CSS, if you decide you need to add a rule to `layout`, you can simply do this:

```
@layer layout {
    .my-Thing {
        display: grid;
        grid-auto-flow: column;
    }
}
```

The rule will be added to the relevant layer and obey the priority assigned to that layer when it was defined.

The priority of rules in and outside of layers

A helpful feature of cascade layers is that any styles outside of a layer are always a greater priority than styles in one. This makes it helpful if you want to, effectively, quarantine styles. This is exactly the procedure we followed at the outset, moving our mythical "Shoe Lace" framework CSS into its own layer, allowing us to write styles for our button easily without needing to battle specificity.

The great thing with cascade layers is that you know that what is left outside any layers will be unaffected. So, perhaps another helpful way to think of cascade layers is as a way to de-emphasize styles by grouping rules into tiers.

Nesting layers inside other layers

If you find yourself in a suitably complicated scenario, you can even nest layers inside other layers. Let's suppose we have layouts split into areas such as `header`, `sidebars`, `content`, and `modals`. You might decide these concerns would be better organized under a `layout` layer. We could set that up like this:

```
@layer framework;
@layer layout {
    @layer header,
            sidebars,
            content,
            modals;
}
@layer components;
```

Then, we use dot notation to add styles to the nested layers like this:

```
@layer layout.header {
    header {
        display: flex;
    }
}
```

We give the outer layer name, then a period character, and then the nested layer name.

Or, if we wanted to `@import` into a nested layer, it would look like this:

```
@import url("header.css") layer(layout.header);
```

The top-to-bottom prioritizing of layers is applied to the nested layers too, so in this instance, `layout.modals` would have a higher priority than `layout.sidebars`, for example. It isn't any more "specific," though, just because it is nested. So, something in `layout.modals` wouldn't have greater priority than something in `components`, for example.

The importance of !important

I feel the use of `!important` has been a little vilified over the years. Like all of the best villains, that's perhaps because it is a little misunderstood. I'll exemplify. Can you explain to me what `!important` does here?

```
/* Author stylesheet */
body {
    font-size: 1.1em !important;
}
```

I mean, sure, we probably all know that if you are having trouble getting a rule to have a certain declaration applied, slapping `!important` on there usually fixes it. But how and why does it actually do that?

Confession. I've been writing CSS for over 20 years, and it was only when reading the specification for cascade layers that I think I fully, truly, understood the mechanics of `!important`. Want to bore fellow developers next time you are sharing a coffee? Pay attention to this next bit; it will be `!important` (see what I did there?).

When styles are applied by the browser onto the page, there is an order of priority:

- Firstly, the browser applies its own set of styles; you often see these marked in your dev tools as **User Agent Style Sheet**.
- Then, user style sheets are applied. You may have none set, but if you were to set a minimum font size in your browser preferences, that setting would go into your user style sheet.
- Finally, the greatest priority of the three is the author style sheets – the sheets you and I write day to day.

With that initial order of precedence established, we can envisage that if `font-size` for the body was set by the browser, the user, and the author, the author styles (that we as developers write) would "win" and take precedence. This works well enough for the most part, but what if a user needs a large minimum `font-size` for accessibility or preference? And, in our ignorance, we thought that 12px was absolutely fine for all? That would be a problem for many! The solution to this issue is the `!important` declaration.

The `!important` declaration inverts the established order. So, not only will an `!important` declaration take precedence over a normal declaration in the same style sheet, but if an `!important` declaration is added to a user style sheet, such as for our aforementioned `font-size` preference, this will take precedence over any author style sheet declaration. This way, users have a guaranteed mechanism to ensure their needs are applied.

In addition, any user-agent (e.g., browser) `!important` declarations trump even the user `!important` styles. There is further nuance if needing to factor in encapsulation contexts such as the shadow trees, but for our purposes, this mental model will suffice.

> The specification for cascade layers offers additional information regarding encapsulation: `https://www.w3.org/TR/css-cascade-5/#cascade-context`.
>
> In addition, Level 6 of the CSS Cascade specification also introduces the notion of scoped styles, which are sure to make isolating groups of rules even more flexible. You can read that draft specification here: `https://www.w3.org/TR/css-cascade-6/#scoped-styles`.

So, what does this have to do with cascade layers?

!important in cascade layers

Let's extend our earlier, simplistic example:

```
@layer reset,
framework,
layout,
components;
```

In this instance, as we have established, in terms of the layers, `reset` will have the least priority, and `components` the most. However, should we add `!important` to one of the declarations in the `reset` layer, this is what would effectively happen:

```
reset,
framework,
layout,
components;
!components
!layout
!framework,
!reset,
```

The browser effectively makes important "versions" of each layer, crucially in reverse order. The outcome of this, practically, is that an `!important` rule in `reset` becomes the highest priority, despite normal rules being the least priority in the "normal" `reset` layer.

So, all this is to help you understand that when using cascade layers, you need to be even more vigilant in your use of `!important`. If you can avoid it in your layers, so much the better, and the more predictable they will be.

As an exercise, take a look at the CSS of `example_06-09`:

```
@layer reset,
type;

@layer reset {
    .para {
        font-size: 2rem;
        color: hotpink !important;
        text-decoration-style: dotted;
        text-decoration: line-through orange !important;
    }
}

@layer type {
    .para {
        font-size: 1.2rem !important;
        color: goldenrod !important;
        text-decoration: solid overline lime !important;
    }
}

.para {
    font-size: 1rem;
    color: brown;
    text-decoration: wavy underline lime !important;
}
```

Before you open up this page in a browser and see which styles get applied, consider what you have just learned. What do you think will be the applied `font-size`, `color`, and `text-decoration`?

Old browsers

All modern browsers have shipped cascade layers, but you can always check support at `https://caniuse.com/css-cascade-layers`.

Unfortunately, there is no way of providing any kind of fallback for cascade layers in older browsers; the layers will simply be ignored. You can test for cascade layers with `@supports`, which we cover in more detail later in this chapter. That would look like this:

```
@supports not at-rule(@layer) {
    /* Styles added here for browsers that don't support layers */
}
```

That concludes our overview of cascade layers.

Selectors, units, and capabilities

Although they may not seem like the most exciting of subjects, selectors, units, and capabilities are the “meat and potatoes” of CSS. Master these and your power to solve problems with CSS will increase substantially. So, skip this section at your peril!

Anatomy of a CSS rule

Before exploring some of the recent additions to CSS, to prevent confusion, let’s establish the terminology we use to describe a CSS rule. Consider the following example:

```
.selector {
    /* comment */
    property: value; /* declaration */
}
```

This rule is made up of the selector (`.selector`) and then, inside the curly braces, the declaration. The declaration is further defined by the property and the value. Happy we’re on the same page? Great, let’s press on.

Pseudo-elements and pseudo-classes

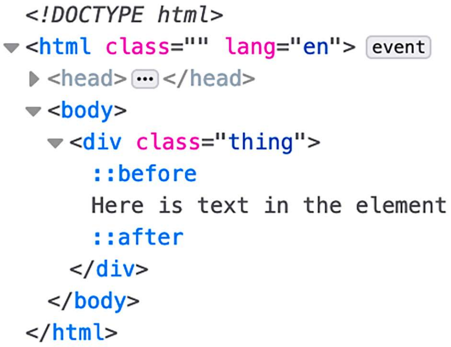
There is potential for some confusion when we go on, shortly, to talk about “pseudo” classes. That is because, in CSS, there are both pseudo-classes and pseudo-elements. Let’s therefore take a moment to establish the difference.

The word “pseudo” in this context means something that is *like* something, but not really it. So, a pseudo-element is something that is like an element but not really one, and a pseudo-class is something that selects something that isn’t really something, but rather the *state* of something. So, `:hover` is a pseudo-state, as it selects something when in a certain state. It’s possible I’m starting to sound like the Riddler now from Batman! Let’s clarify with some code. Here is how you create a pseudo-element in CSS:

```
.thing::before {
    content: "Spooky";
}
```

That inserts a `::before` pseudo-element into the `.thing` element with the content/text `"Spooky"`. A `::before` behaves like a first child of the element it is defined on, and an `::after` behaves like the last child of the element.

The following figure might help. It shows a single element represented in the Firefox developer tools, containing text with both a `::before` and an `::after` pseudo-element added in CSS:



```
<!DOCTYPE html>
<html class="" lang="en"> event
  <head>⋯</head>
  <body>
    <div class="thing">
      ::before
      Here is text in the element
      ::after
    </div>
  </body>
</html>
```

Figure 6.1: Developer tools will show you where pseudo-elements are in the DOM

The key thing to remember with pseudo-elements is that if you don't provide a value for `content`, nothing will show on the page. Notice the double colon before `before`? Officially, that is how you should code pseudo-elements as it helps differentiate them from pseudo-classes, which only use one. However, a single colon worked with the first implementations of `::before` and `::after`, so you can still write them that way too.

You can't do the same with pseudo-classes; they always have a single colon. For example, `:hover`, `:active`, and `:focus` are all pseudo-classes and are written with a single colon.

Hopefully, the difference between pseudo-classes and elements is now clear.

With that distinction made, let's move on and look at some of the powerful selectors available to us today in CSS.

CSS selectors – beyond the normal!

CSS now provides incredible power for selecting elements within a page. If you are only used to selecting elements based on their class, ID, or element type, these techniques may open your eyes to new possibilities. I'd better qualify that bold claim.

CSS attribute selectors

You've probably used CSS attribute selectors to create rules. For example, consider the following markup:

```
<img src="https://placeimg.com/640/480/any" alt="an inquisitive cat" />
```

Also consider this CSS:

```
img[alt] {
    border: 3px dashed #e15f5f;
}
```

This would select the `img` element in the preceding code, and any others on the page, provided that they have an `alt` attribute.

In fact, to make something a little more useful, we could combine this with the `:not` negation selector (we will look at that in detail later in this chapter) to add a red border around any images that have no `alt` attribute or an `alt` attribute with no value:

```
img:not([alt]),
img[alt=""] {
    border: 3px solid red;
}
```

That would be useful for testing accessibility, as it would visually highlight any images that didn't have alternate text included for assistive technology.

As another example, let's say we wanted to select all elements with a `data-sausage` attribute:

```
[data-sausage] {
    /* styles */
}
```

The key thing here is to use square brackets to specify the attribute you want to select.

The `data-` type attribute was introduced in HTML5 to provide a place for custom data that can't be stored sensibly by any other existing mechanism. The specification description for these can be found here: `https://www.w3.org/TR/2010/WD-html5-20101019/elements.html#embedding-custom-non-visible-data-with-the-data-attributes`.

You can also narrow things down by specifying what the attribute value is. For example, consider the following rule:

```
img[alt="Sausages cooking"] {
    /* Styles */
}
```

This would only target images that have an `alt` attribute value of `"Sausages cooking"`. Here's an example:

```
<img src="img/sausages.png" alt="Sausages cooking" />
```

So far, so "big deal, what else you got, CSS guy?" I hear you. Let's turn things up a notch, shall we?

CSS substring matching attribute selectors

We also have the ability to select elements based on the substring of their attribute selector. That sounds complicated. It isn't! The three options are whether the attribute does the following:

- Begins with a certain substring
- Contains an instance of a certain substring
- Ends with a certain substring

Let's see what they look like.

The "beginning with" substring matching attribute selector

Consider the following markup:

```
<li data-type="todo-chore">Empty the bins</li>
<li data-type="todo-exercise">Play football</li>
```

Suppose that markup represents two items in a "to-do" list application we are building. Even though they both have different `data-type` attribute values, we can select them both with the "beginning with" substring matching attribute selector, like this:

```
[data-type^="todo"] {
    /* Styles */
}
```

The key character in all this is the ^ symbol. That symbol is called the **caret**, although it is often referred to as the **hat** symbol too. In this instance, it signifies "begins with." Because both `data-type` attributes have values that begin with `"todo"`, our selector selects them.

The “contains an instance of” substring matching attribute selector

The “contains an instance of” substring matching attribute selector has the following syntax:

```
[attribute*="value"] {
    /* Styles */
}
```

Like all attribute selectors, you can combine them with a type selector (one that references the actual HTML element used) if needed, although I would only do that if I had to—in case you want to change the type of element used. Let’s try an example. Consider this markup:

```
<p data-ingredients="scones cream jam">Will I get selected?</p>
```

We can select that element like this:

```
[data-ingredients*="cream"] {
    color: red;
}
```

The key character in all this is the `*` (asterisk) symbol, which in this context means “contains.” The “begins with” selector would not have worked in this markup, as the string inside the attribute didn’t begin with `"cream"`. It did, however, contain `"cream"`, so the “contains an instance of” substring attribute selector finds it.

The “ends with” substring matching attribute selector

The “ends with” substring matching attribute selector has the following syntax:

```
[attribute$="value"] {
    /* Styles */
}
```

An example should help. Consider this markup:

```
<p data-ingredients="scones cream jam">Will I get selected?</p>
<p data-ingredients="toast jam butter">Will I get selected?</p>
<p data-ingredients="jam toast butter">Will I get selected?</p>
```

Suppose we only want to select the element with `"jam"` at the end of the `data-ingredients` attribute (which would be the first element of those three). We can’t use the “contains an instance of” (it will select all three) or “begins with” (it will only select the last one) substring attribute selector. However, we can use the “ends with” substring attribute selector:

```
[data-ingredients$="jam"] {
    color: red;
}
```

The key character in all this is the `$` (dollar) symbol, which means “ends with.”

Right, we have some pretty handy attribute-related selectors now. But it’s also worth knowing that you can chain attribute selectors, just like you can class selectors.

Chaining attribute selectors

You can have even more possibilities for selecting items by grouping attribute selectors. Suppose we had this markup:

```
<li
    data-todo-type="exercise"
    data-activity-name="running"
    data-location="indoor"
>
    Running
</li>
<li
    data-todo-type="exercise"
    data-activity-name="swimming"
    data-location="indoor"
>
    Swimming
</li>
<li
    data-todo-type="exercise"
    data-activity-name="cycling"
    data-location="outdoor"
>
    Cycling
</li>
<li
    data-todo-type="exercise"
    data-activity-name="swimming"
    data-location="outdoor"
>
    Swimming
</li>
```

Let's suppose I only wanted to select "indoor swimming." I can't use just `data-location="indoor"` as that would get the first element too. I can't use `data-activity-name="swimming"` as that would get me the second and the fourth. But, I can do this:

```
[data-activity-name="swimming"][data-location="indoor"] {
    /* styles */
}
```

This selects elements that have `"swimming"` as the activity name, as long as they also have `"indoor"` as the location.

> Attribute selectors allow you to select IDs and classes that start with numbers. Before HTML5, it wasn't valid markup to start IDs or class names with a number. HTML5 removes that restriction. When it comes to IDs, there are still some things to remember. There should be no spaces in the ID name, and it must be unique on the page. For more information, visit `https://html.spec.whatwg.org/multipage/dom.html#the-id-attribute`. Now, although you can start ID and class values with numbers in HTML5, CSS still restricts you from using ID and class selectors that start with a number (`https://www.w3.org/TR/CSS21/syndata.html#characters`). Luckily for us, we can easily work around this by using an attribute selector; for example, `[id="10"]`.

Right, I think our attribute-selecting skills are now pretty good. Let's move on to how we can deal with selecting elements based on where they are in the document.

CSS structural pseudo-classes

CSS gives us more power to select elements based on where they sit in the structure of the DOM. Let's consider a common design treatment; we're working on the navigation bar for a larger viewport and we want to have all but the last link over on the left. Historically, we would have needed to solve this problem by adding a class name to the last link so we could select it, like this:

```
<nav class="nav-Wrapper">
    <a href="/home" class="nav-Link">Home</a>
    <a href="/About" class="nav-Link">About</a>
    <a href="/Films" class="nav-Link">Films</a>
    <a href="/Forum" class="nav-Link">Forum</a>
    <a href="/Contact-Us" class="nav-Link nav-LinkLast">Contact Us</a>
</nav>
```

This in itself can be problematic. For example, sometimes, just getting a content management system to add a class to a final list item can be frustratingly difficult. Thankfully, in those eventualities, we can solve this problem, and many more, with CSS structural pseudo-classes.

The :last-child selector

Way back in CSS 2.1, we already had a selector applicable for the first item in a list – the `:first-child` selector:

```
div:first-child {
    /* Styles */
}
```

However, CSS Level 3 added a selector that can also match the last:

```
div:last-child {
    /* Styles */
}
```

Let's look at how that selector could fix our prior problem if we didn't want to, or couldn't, add another class at the desired point in the markup:

```
.nav-Wrapper {
    display: flex;
}
.nav-Link:last-child {
    margin-left: auto;
}
```

There are also useful selectors for when something is the only item (`:only-child`) and the only item of a type (`:only-of-type`).

The nth-child selectors

The `nth-child` selectors let us solve even more difficult problems. With the same markup as before, let's consider how `nth-child` selectors allow us to select any arbitrary link(s) we want within the list.

Firstly, what about selecting every other list item? We could select the odd ones like this:

```
.nav-Link:nth-child(odd) {
    /* Styles */
}
```

Or, if you wanted to select the even ones, you could do this:

```
.nav-Link:nth-child(even) {
    /* Styles */
}
```

Understanding what nth rules do

For the uninitiated, nth-based selectors can look pretty intimidating. However, once you've mastered the logic and syntax, you'll be amazed at what you can do with them. Let's take a look.

Here are the `nth`-based selectors at our disposal:

- `nth-child(n)`
- `nth-last-child(n)`
- `nth-of-type(n)`
- `nth-last-of-type(n)`

We've seen that we can use `(odd)` or `(even)` values already in an `nth`-based expression, but the `(n)` parameter can be used in another couple of ways:

- As an integer – for example, `:nth-child(2)` would select the second item. Passing a number/integer into the nth selector is easy enough to understand; just enter the element number you want to select.
- However, you can also pass a numeric expression – for example, `:nth-child(3n+1)` would start at the first element and then select every third element.

The numeric expression version of the selector is the part that can be a little baffling at first. Let's break it down.

Breaking down the math

Let's consider 10 spans on a page (you can play about with these by looking at `example_06-02`):

```
<span></span>
<span></span>
<span></span>
<span></span>
<span></span>
<span></span>
<span></span>
<span></span>
<span></span>
<span></span>
```

We will style them like this:

```
span {
    height: 2rem;
    width: 2rem;
    background-color: blue;
    display: inline-block;
}
```

As you might imagine, this gives us 10 squares in a line:

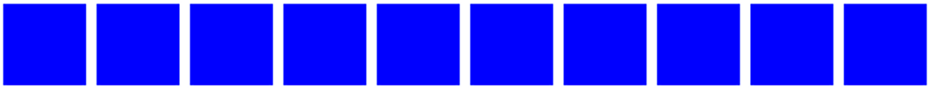


Figure 6.2: We will test our nth-child selection skills on these 10 identical elements

OK, let's look at how we can select different ones with `nth`-based selections.

For practicality, when considering the expression within the parentheses, I start from the right. So, for example, if I want to figure out what `(2n+3)` will select, I start with the right-most number (the "3" here indicates the third item from the left) and know it will select every second element from that point on. So, say we add this rule:

```
span:nth-child(2n + 3) {
    background-color: #f90;
    border-radius: 50%;
}
```

This results in this in the browser:

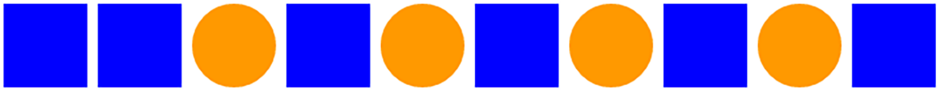


Figure 6.3: Anything that matches our nth-child selector is turned round and orange

As you can see, our `nth` selector targets the third list item and then every subsequent second one after that too. If there were 100 list items, it would continue selecting every second one.

How about selecting everything from the second item onward? Well, although you could write `:nth-child(1n+2)`, you don't actually need the first number 1 as, unless otherwise stated, `n` is equal to 1. We can, therefore, just write `:nth-child(n+2)`. Likewise, if we wanted to select every third element, rather than writing `:nth-child(3n+3)`, we could just write `:nth-child(3n)` as every third item would begin at the third item anyway, without needing to state it explicitly.

The expression can also use negative numbers; for example, `:nth-child(3n-2)` starts at minus 2 and then selects every third item.

You can also change the direction. By default, once the first part of the selection is found, the subsequent ones go down the elements in the DOM (and therefore from left to right in our example). However, you can reverse that with a minus; for example:

```
span:nth-child(-2n + 3) {
    background-color: #f90;
    border-radius: 50%;
}
```

This example finds the third item again, but then goes in the opposite direction to select every two elements – up the DOM tree, and therefore from right to left in our example:

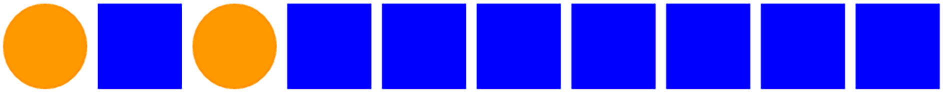


Figure 6.4: With a minus symbol, we can select in the opposite direction

Hopefully, the `nth`-based expressions are making more sense now. The `nth-child` and `nth-last-child` expressions differ in that the `nth-last-child` variant works from the opposite end of the document tree. For example, `:nth-last-child(-n+3)` starts at 3 from the end and then selects all the items after it. Here's what that rule gives us in the browser:

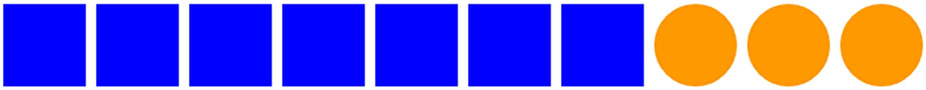


Figure 6.5: nth-last-child lets you start from the opposite end of the elements

Let's now consider `:nth-of-type` and `:nth-last-of-type`. While the previous examples count any children, regardless of type (always remember the `nth-child` selector targets all children at the same DOM level, regardless of classes), `:nth-of-type` and `:nth-last-of-type` let you be specific about the type of item you want to select. Consider the following markup (`example_06-03`), which is a mixture of `div` and `span` elements, albeit with the same class:

```
<span class="span-class"></span>
<span class="span-class"></span>
<span class="span-class"></span>
<span class="span-class"></span>
<span class="span-class"></span>

<div class="span-class"></div>
<div class="span-class"></div>
<div class="span-class"></div>
<div class="span-class"></div>
<div class="span-class"></div>
```

Let's say we used this selector:

```
.span-class:nth-of-type(-2n + 3) {
    background-color: #f90;
    border-radius: 50%;
}
```

Even though all the elements have the same `span-class` class, they don't get seen as one group. The selector applies once to the `span` elements, and then to the `div` elements. Here is what gets selected:

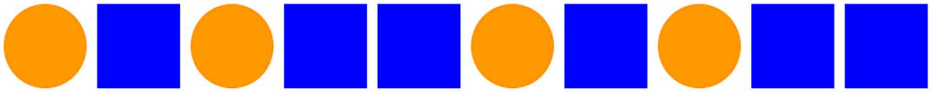


Figure 6.6: nth-of-type selectors work on each type of element they find

CSS doesn't count like JavaScript! If you're used to using JavaScript, you'll know that it counts from 0 upward (zero-index-based). For example, if selecting an element in JavaScript, an integer value of 1 would actually be the second element. CSS, however, starts at 1, so that a value of 1 is the first item it matches.

The of selector

Just before we jump into a more challenging example of how to apply complex `nth`-based selectors, I want to tell you about the "of selector." This allows you to combine `nth-child` rules with further filtering based on the selector. Consider this markup:

```
<span class="square"></span>
<span class="square"></span>
<span class="circle"></span>
<span class="square"></span>
<span class="circle"></span>
<span class="square"></span>
<span class="circle"></span>
<span class="square"></span>
<span class="circle"></span>
<span class="square"></span>
<span class="square"></span>
```

Also consider this CSS:

```
.square {
    display: inline-block;
    height: 2rem;
    width: 2rem;
    background-color: blue;
}

.circle {
    display: inline-block;
    height: 2rem;
    width: 2rem;
    background-color: #f90;
    border-radius: 50%;
}

:nth-child(2n of span.circle) {
    background-color: red;
}
```

The first two rules are just setting up the default shapes and colors – blue squares and red circles. The thing to focus on is the last rule, which is causing two of the circles to be red.

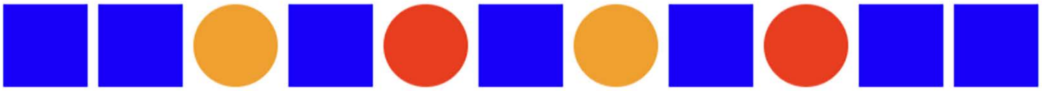


Figure 6.7: The of selector syntax allows fine-grained control

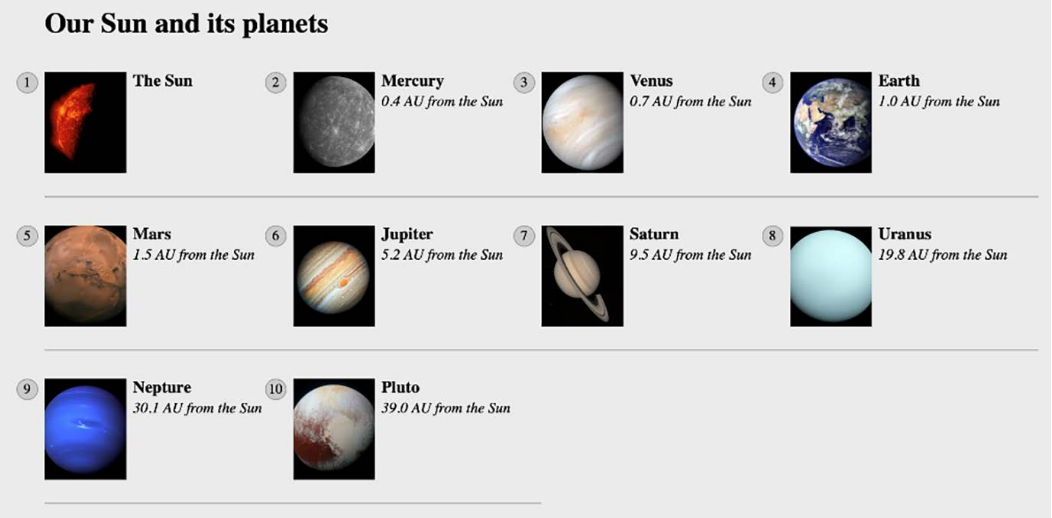
Given what you know about `nth` selectors, does that syntax make sense? We are selecting every second child, as long as it is a span element with a class of circle.

You can play about with that code in `example_06-04`.

nth-based selection in responsive web designs

Just to close out this little section, I want to illustrate a real-life responsive web design problem and how we can use nth-based selection to solve it. Imagine we are building a page showing our sun and the various planets of our solar system. Our content management system simply spits all the items out in a list, but we want to show them in a grid of sorts.

For some viewports, we will only be able to make it two items wide. However, as the viewport increases in width, we can show three items, and at larger sizes still, we can show four. Here is the problem, though. Regardless of the viewport size, we want to prevent any items on the bottom row from having a border on the bottom. You can view this code in `example_06-05`. Here is how it looks when it's four items wide:



Our Sun and its planets

1. **The Sun**
2. **Mercury** *0.4 AU from the Sun*
3. **Venus** *0.7 AU from the Sun*
4. **Earth** *1.0 AU from the Sun*
5. **Mars** *1.5 AU from the Sun*
6. **Jupiter** *5.2 AU from the Sun*
7. **Saturn** *9.5 AU from the Sun*
8. **Uranus** *19.8 AU from the Sun*
9. **Nepture** *30.1 AU from the Sun*
10. **Pluto** *39.0 AU from the Sun*

Figure 6.8: Our task here is to remove the border on the bottom row, regardless of how many items are showing

See that pesky border below the bottom two items? That's what we need to remove. However, I want a robust solution so that if there were another item on the bottom row (if another planet is suddenly discovered hiding behind Uranus!), the border would also be removed from that.

Now, because there is a different number of items on each row at different viewports, we will also need to change the nth-based selection at different viewport widths. But we can use media queries for that. For the sake of brevity, I'm not going to show you the selection for each media query. I'll just show you the selection that matches four items per row, as you can see in the preceding screenshot. However, you can open the code sample to see the selections for each different viewport:

```
@media (min-width: 55rem) {
    .Item {
        width: 25%;
    }
    /* Get me every fourth item and, of those, only ones that are in the
    last four items */
    .Item:nth-child(4n + 1):nth-last-child(-n + 4),
    /* Now get me every one after that same collection too. */
    .Item:nth-child(4n + 1):nth-last-child(-n + 4) ~ .Item {
        border-bottom: 0;
    }
}
```

You'll notice here that we are chaining the nth-based pseudo-class selectors, much like we chained attribute selectors earlier in this chapter. It's important to understand when you chain nth selectors like this that the first selector doesn't "filter" the selection for the next selector; rather, the element has to match each of the selections.

So, let's take a line like this:

```
.Item:nth-child(4n + 1):nth-last-child(-n + 4),
```

`.Item` has to be the first item of four and also be one of the last four:

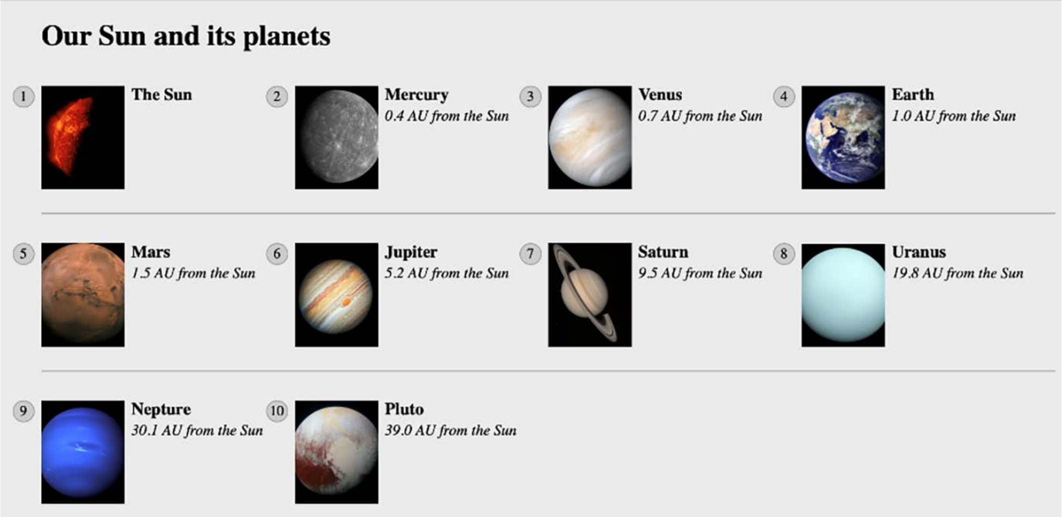

Our Sun and its planets
1
The Sun
2
Mercury
0.4 AU from the Sun
3
Venus
0.7 AU from the Sun
4
Earth
1.0 AU from the Sun
5
Mars
1.5 AU from the Sun
6
Jupiter
5.2 AU from the Sun
7
Saturn
9.5 AU from the Sun
8
Uranus
19.8 AU from the Sun
9
Nepture
30.1 AU from the Sun
10
Pluto
39.0 AU from the Sun

Figure 6.9: Achievement unlocked—using nth-child selectors like some kind of nth-child wizard!

Nice! Thanks to `nth`-based selections, we have a defensive set of rules to remove the bottom border, regardless of the viewport size or the number of items we are showing. You can view the completed code in `example_06-06`.

Now, one nifty bit of selection we are doing in that prior example is using the "subsequent sibling" selector. We haven't looked at that, so we will cover it next.

Combinator selectors – child, next sibling, and subsequent sibling

Sometimes you want to do something particular to an element, but only if it follows another, or perhaps only if it is the child of another element. We are going to look now at how we can do that.

I'm making the assumption at this point that you understand the basic selector pattern, where one class followed by another selects any descendant that matches. For example, `.parent .descendant {}` would select any element that was a descendant of the `.parent` element with a class of `.descendant`, no matter how many levels deep.

The child combinator

The child combinator only selects direct descendants. Consider this markup:

```
<div class="parent">
    <div class="descendant child">
        <div class="descendant grandchild"></div>
    </div>
</div>
```

We can select only the direct "child" of the parent element, like this:

```
.parent > .descendant {
    /* Styles */
}
```

Notice the right-angle bracket between the two class names in the selector; that's the child combinator symbol.

The next sibling

Consider another example:

```
<div class="item one">one</div>
<div class="item">two</div>
<div class="item">three</div>
<div class="item">four</div>
<div class="item">five</div>
<div class="item">six</div>
```

Let's suppose we wanted to select an item, but only if it is the next sibling of `.one`. With that markup, we could select the second element in that list like this:

```
.one + .item {
    border: 3px dashed #f90;
}
```

The + symbol there means "next sibling," so select the next sibling element of `.one` – our second element.

The subsequent sibling

With the same markup from the previous example, if we wanted to select all items after the third, we could do this:

```
.item:nth-child(3) ~ .item {
    border: 3px dashed #f90;
}
```

The ~ symbol, called a **tilde**, says "every subsequent sibling."

The negation (:not) selector

Another handy selector is the negation pseudo-class selector. This is used to select everything that isn't something else. Consider this:

```
<div class="a-div"></div>
<div class="a-div"></div>
<div class="a-div"></div>
<div class="a-div not-me"></div>
<div class="a-div"></div>
```

Then, consider these styles:

```
.a-div {
    display: inline-block;
    height: 2rem;
    width: 2rem;
    background-color: blue;
}
.a-div:not(.not-me) {
    background-color: orange;
    border-radius: 50%;
}
```

Our final rule will make every element with a class of `.a-div` orange and round, with the exception of the `div` that also has the `.not-me` class. You can find that code in the `example_06-05` folder of the code samples (remember, you can grab them all at `https://rwd.education`):

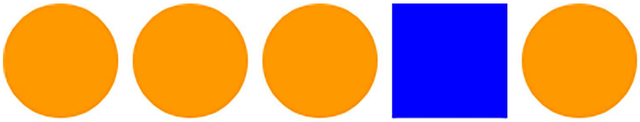


Figure 6.10: The negation selector allows you to exclude elements from selection

So far, we have looked primarily at what's known as structural pseudo-classes (full information on this is available at `https://www.w3.org/TR/selectors/#structural-pseudos`). However, CSS has many more selectors. If you're working on a web application, it's worth looking at the full list of UI element state pseudo-classes (`https://www.w3.org/TR/selectors/#UIstates`) as they can, for example, help you target rules based on whether something is selected or not.

The empty (:empty) selector

I've encountered situations where I have an element that includes some padding on the inside and gets content dynamically inserted. Sometimes, it gets content inserted, while sometimes it doesn't. The trouble is, when it doesn't include content, I still see the padding. Consider the HTML in `example_06-06`:

```
<div class="thing"></div>
```

Here's the CSS:

```
.thing {
    padding: 1rem;
    background-color: violet;
}
```

Without any content in that element, I still see the background color. Thankfully, we can easily hide it, like this:

```
.thing:empty {
    display: none;
}
```

However, just be careful with the `:empty` selector. For example, you might think this is empty:

```
<div class="thing"> </div>
```

It isn't! Look at the whitespace in there. Whitespace is not "no" space! And neither is something like a line break.

However, just to confuse matters, be aware that a comment doesn't affect whether an element is considered "empty" or not. For example, this is still considered empty:

```
<div class="thing"><!--I'm empty, honest I am--></div>
```

Grouping selectors

Ordinarily, when we want to share a bunch of properties across a number of things, we just comma-separate the selectors into a list:

```
.thing1,
[data-thing],
#thing {
    /* styles */
}
```

That works fine for a few things. All those comma-separated selectors get the same styles applied, but it can start to become unwieldy and repetitive in more complicated scenarios:

```
.blog-post > h1,
.blog-post > h2,
.blog-post > h3,
.blog-post > h4,
.blog-post > ul,
.blog-post > ul li,
.blog-post > p,
.blog-post > a {
    /* shared styles */
}
```

The other downside is that a single mistake in that group renders the whole list invalid:

```
.blog-post > h1,
.blog-post > h2,
.blog-post > h3,
.blog-post > h4,
.blog-post > ul,
.blog-post > ul < li,
.blog-post > p,
.blog-post > a {
    /* shared styles */
}
```

That accidental "less than" symbol will mean this rule won't apply any styles to any of the selectors. Now, I don't think that is something to concern yourself with too much, but there are newer selectors that can solve these issues.

The :is() functional pseudo-class

This is also known as the "matches any" selector. Let's use it to refactor that prior example, and then talk about the differences:

```
.blog-post > :is(h1, h2, h3, h4, ul, ul < li, p, a) {
    /* shared styles */
}
```

Not only is this more compact, but it has the added benefit that even though that invalid selector is still present, the entire rule is not invalidated; just that one. Nice, eh?

Now, it is important to be aware that the `:is` functional gives all selectors within the specificity of the most specific selector. That can be problematic. Here is the relevant portion of the specification:

> *The specificity of an* `:is()`*,* `:not()`*, or* `:has()` *pseudo-class is replaced by the specificity of the most specific complex selector in its selector list argument.*

So, suppose you have a grouping like this:

```
:is(.thing, .another-thing, .and-another) {
    /* styles */
}
```

Each of those selectors will only have a specificity of 0,1,0, as the most specific selector in the selector list is a class.

However, let's say I add an ID selector in there:

```
:is(#ouch .thing, .another-thing, .and-another) {
    /* styles */
}
```

Each one of those now gets the specificity of ID: 1,0,0. That can introduce problems in the long term, as it means you have a very high specificity selector in your code base, and should you ever need to override it at some point, you will need to create a rule with even greater specificity. As such, it is generally advisable to avoid using IDs unless absolutely necessary.

For more on specificity and the maintenance of larger CSS code bases, you may be interested in another of your author's books, *Enduring CSS*!

However, if you do find yourself in such a pickle, we have just the selector for you.

The :where() functional pseudo-class

The `:where()` selector takes a selector list, just as `:is()` does. However, this selector is snappily known as the "specificity-adjustment pseudo-class." Cripes!

Any selector inside the `:where()` selector list is given a specificity of zero – even big bad ID selectors! So, if you do find yourself in a scenario where you need to use an ID selector, you can wrap it in `:where()` and it will not add to specificity. Let's take a look.

Suppose we had markup like this:

```
<div id="thing" class="hi">hi</div>
```

Let's say we added CSS like this:

```
.hi {
    width: 200px;
    height: 200px;
    background-color: gold;
}

#thing {
    background-color: red;
}
```

You can hopefully guess that the square we have will be red. The ID is an order of magnitude more specific than the class, and it is also written after it in the stylesheet – so no way that square is going to have a gold background! However, wrap it in a `:where()`:

```
.hi {
    width: 200px;
    height: 200px;
    background-color: gold;
}

:where(#thing) {
    background-color: red;
}
```

That ID has zero specificity, meaning our square displays gold. This selector is useful in lots of scenarios, but I find it most useful for "reset" styles, or any style you know you'll be overriding later on.

> Remember to check support for your users. As we delve into CSS more and more, don't forget to visit `https://caniuse.com`, if you ever want to know what the current level of browser support is for a particular CSS or HTML5 feature. Alongside showing browser version support (searchable by feature), it also provides the most recent set of global usage statistics from `https://gs.statcounter.com`.

The :has() relational pseudo-class

A very recent addition to the CSS selectors at our disposal is the `:has()` functional pseudo-class.

In simple terms, it's a way of selecting an element, as long as it has certain child elements or conditions.

Let's suppose you want to do something with a container, but only if it has a `video` element inside. That would look like this:

```
.thing:has(video) {
    /* Styles */
}
```

Or, you want to add some padding in the block flow direction if an element doesn't have a `p` element inside of it. That would look like this:

```
.other-thing:not(:has(p)) {
    padding-block: 10px;
}
```

How about an even more convoluted use case? Suppose you want to select an `h1` if there is an `img` directly after it. We can do that like this:

```
h1:has(+ img) {
    /* Styles */
}
```

There are so many possibilities with `:has()`!

If you need to fork your code based on support for `:has()`, you can do this with feature queries. That would look like this:

```
@supports selector(:has(*)) {
    /* Styles */
}
```

Now, don't worry if you've not come across feature queries before; we are going to look at them later on in this very chapter.

Logical Properties and Values

You may have noticed in one of the previous examples that I used the `padding-block` shorthand. That's part of the Logical Properties and Values specification and allows us to define spacing and more based on the logical direction of the document. This can be useful when the writing mode being used might not match the physical mappings. For example, in some languages, you start reading from the right and not the left. If you use a logical property in that instance, instead of an absolute one, it will work as intended in either case.

Practically, as I don't need to worry about writing modes too often, I find these properties most useful for setting padding or margin on an element in two places at once. For example, if my document is flowing from top to bottom, I can set a margin on either side of an element with `margin-inline`, which I find preferable to setting it with the `margin` shorthand, where I would need to set top and bottom and left and right each time..

The specification for these properties can be read at `https://www.w3.org/TR/css-logical-1/`.

Limiting the application of selectors with @scope

A fairly recent addition, and not yet widely supported feature of CSS, is the ability to isolate the application of selectors based on a range in the DOM, and do so with less specificity. We do this with the `@scope` at-rule.

Just like all the other at-rules in CSS, it is a way to wrap a bunch of rules to apply them only in certain circumstances.

Practically, I think of `@scope` as a means to select the same things but with less specificity. I'll explain the specificity side of things momentarily. First, let's consider how to use scope with an example. Consider this DOM structure, which can be found in `example_06-11`.

```
<section class="rwd-Chapters">
    <h2>
        Talking about the
        <code>@scope</code>
        at-rule
    </h2>
    <ol>
        <li>
            <p>
                The
                <code>@scope</code>
                at rule can de-specify selection of elements.
            </p>
        </li>
    </ol>
</section>
```

Consider if we wanted to target `code` elements that were descendants of `p` tags inside `li` tags that were descendants of the section with a class of `.rwd-Chapters`.

To style that, we might write a selector like this:

```
.rwd-Chapters li p code {}
```

We didn't use an ID to target anything, but as there is a class and 3 elements in that selector, it gives a specificity score of 0, 1, 3. That's a relatively specific selector, which is going to make it harder to override with a different style if needed. To do so, we would need another selector with at least a class and 3 elements further down the style sheet, as just a single class would not be enough.

> You can test the specificity of your CSS selectors with a number of online tools. I quite like `https://polypane.app/css-specificity-calculator/`. However, most modern developer tools will also show you the specificity of a selector if you hover over it in the styles panel.
>
> Lower specificity is generally preferred on large code bases or, perhaps more accurately, uniformly low specificity selectors.

Targeting elements with @scope to get less specificity

Let's look at how we could target the same elements using `@scope`. We pass `@scope` either a beginning and end point, just an end point, or just a beginning. In this case, we need only specify the beginning, known as the **scoping root**:

```
@scope (.rwd-Chapters li p) {
    code {
    }
}
```

This targets the relevant `code` element, excluding ones in the `h2` tag, exactly as before, but crucially only has a specificity of 0,0,1 because that is the selector used inside `@scope` – more on that in a moment.

If you specify just an endpoint, then the rule is going to flow down from the top of the DOM, selecting any `code` elements unless they are inside an `h2` inside an element with a class of `.rwd-Chapters`.

```
@scope to (.rwd-Chapters h2) {
    code {
        background-color: #555;
        color: #efefef;
        padding: 0.2em 0.5em;
        border-radius: 0.2em;
        font-weight: 600;
    }
}
```

@scope specificity

The selectors used in the scope root and/or limit do not contribute to the specificity of the selector inside. In our prior example, `code` still has a selector specificity of 0,0,1.

However, you can use the associated `:scope` pseudo-class inside of `@scope`, and that adds a specificity of 0,1,0, like any other pseudo-class. We haven't looked at `:scope` yet – it lets you reference the scoping root inside the scoping rule. You can use it to style the root itself, or use it to define another selector relationship. We could use `:scope` like this:

```
@scope (li p) {
    :scope code {
        background-color: #555;
        color: #efefef;
        padding: 0.2em 0.5em;
        border-radius: 0.2em;
        font-weight: 600;
    }
}
```

That means that our `code` selector, prepended by `:scope`, now has a specificity of 0,1,1 (the total of one pseudo-class, plus one element).

So far, so straightforward. However, consider the nesting parent selector `&`. Things get a little more complicated now. Let's suppose we did this instead:

```
@scope (li p) {
    & code {
        background-color: #555;
        color: #efefef;
        padding: 0.2em 0.5em;
        border-radius: 0.2em;
        font-weight: 600;
    }
}
```

In that case, we would end up with the selector only having a specificity of 0,0,3. Why is that? When we use the `&` inside of scope, it gets effectively rewritten behind the scenes as if the parent selector was inside an `:is()` pseudo-class function, like this: `:is(li p) code {}.` As we know from earlier in this chapter, `:is()` takes on the specificity of its most specific argument.

In this instance, it is only element selectors, so the specificity has remained low. But if we used an ID, for example, in the scope root here, the use of `&` would generate a far more specific selector than using the `:scope` pseudo-class.

Using scope inside a style tag

If you apply `@scope` inside a style tag inside the DOM, it is automatically applied to the scope of the DOM it sits within. This pattern – co-locating style tags with markup – is something I commonly see with web apps, where the HTML, CSS, and JavaScript of components are often authored in the same file.

Consider this HTML:

```
<section class="rwd-Chapters">
    <h2>
        <style>
            @scope {
                code {
                    color: red;
                }
            }
        </style>
        Talking about the
        <code>@scope</code>
        at-rule
    </h2>
    <ol>
        <li>
            <p>
                The
                <code>@scope</code>
                at rule can de-specify selection of elements.
            </p>
        </li>
    </ol>
</section>
```

In this case, the `<code>` inside the `h2` element will appear red. Because the `@scope` is placed physically where you want the scope to be, it is not necessary to define the scope root.

The practicality of @scope

I'm going to say that, at the moment, I'm not a huge fan of scope. I appreciate it as a means of selecting elements with less specificity, and there will certainly be occasions where it may prove useful, but I find it too cumbersome, requiring too much cognitive load for casual, everyday use. I also believe in the complex systems that would most benefit this – the problem has been solved more robustly with tooling already.

In its defense, it is new – not even implemented in Firefox as I write this – so perhaps it will grow on me. And it is always good to have abilities in the native web platform, rather than relying on tooling. For now, I still favor simply adding a meaningfully named class to the elements I want to style and just applying the styles that way. Much like `@layer`, there may be specific use cases where `@scope` proves essential, I just haven't found them yet.

Let's change tack now. We've looked at how we can select items in our responsive world. But how about how we size them? Let's take a look at viewport percentage lengths next.

Responsive viewport relative lengths

CSS Values and Units Module Level 3 (`https://www.w3.org/TR/css3-values/#viewport-relative-lengths`) ushered in **viewport relative units**. These are great for responsive web design, as each unit is a percentage length of the viewport:

- The `vw` unit, where each `vw` unit is 1% of the viewport width
- The `vh` unit, where each `vh` unit is 1% of the viewport height
- The `vmin` unit (for viewport minimum; equal to the smaller of either `vw` or `vh`)
- The `vmax` (viewport maximum; equal to the larger of either `vw` or `vh`)

Want a modal window that's 90% of the browser height? This is, at least in theory, as easy as this:

```
.modal {
    height: 90vh;
}
```

Now, as useful as viewport relative units are, some browsers have had curious implementations. Safari in iOS, for example, changes the viewable screen area as you scroll from the top of a page, shrinking the address bar as you go, but crucially doesn't make any changes to the reported viewport height. So, that thing you sized to be 90% of the viewport may no longer be 90% of the viewport once you scroll. That can be extremely frustrating.

As such, there is now a bunch of extra related viewport units (`https://www.w3.org/TR/css-values-4/#viewport-variants`):

- `lvh`/`lvw`, where the `lv` is for **largest viewport** percentage units. So, in our iOS example, imagine the browser in a state where any shrinking UI is out of the way and the viewport is as large as it can be; this is the size that `lvh` would be a percentage of.
- `svh`/`svw` is for **smallest viewport** units. This represents a percentage of the screen on the basis of any UI being at its largest state.
- `dvh`/`dvw` is for **dynamic viewport** units. These represent a percentage of what the browser is currently using, possibly between the largest and smallest. For example, if a browser has two states, largest and smallest, as is the case with iOS Safari currently, this value would choose one or the other. The problem here is twofold. Firstly, the browser doesn't necessarily animate between the two values as they change, so it may be choppy. Secondly, if the browser does decide to animate that value as its UI sizes change, it may not always be preferable. Imagine setting your font size using that unit and then seeing all the text resize as you scroll. Probably not the best user experience!

However, you can perhaps find more utility for these units when coupled with fonts. For example, it's now trivially easy to create text that scales in size, depending on the viewport.

Here's an example:

```css
.hero-text {
    font-size: 10vw;
}
```

Now, the text will always be sized as a percentage of the viewport width. However, the limit with something like `vw` units for text, used in isolation, quickly becomes apparent as you use them. Our example size of `10vw` would be a large headline size of `32px` at a small viewport size of `320px`, but a rather over-the-top size of `130px` at a larger viewport of `1300px`.

We can solve this kind of issue easily with the CSS `clamp` function. But that's a whole other topic, and you will need to head over to *Chapter 13*, *Custom Properties and CSS Functions*, to get the low-down on that.

For newer viewport-related units we just looked at, and many other newer CSS features, it can be really handy to be able to **fork** your code for when those features or selectors are supported and when they are not. For that, we can reach for feature queries. Let's look at how to use those next.

Using @supports to fork CSS

When you're building out a responsive web design, attempting to provide a single design that works everywhere, on every device, it's a simple fact that you'll frequently encounter situations when features or techniques are not supported on certain devices. It might be that you want to do something with a newer selector, such as `:has()`, which we looked at earlier in the chapter. In such an instance, you may want to create a fork in your CSS. If the browser supports a feature, provide one set of rules; if it doesn't, provide different rules.

This is the kind of situation that gets handled by `if/else` or `switch` statements in JavaScript. In CSS, we use the `@supports` at-rule.

Now, before we dig into feature queries, it is worth pointing out a straightforward way of providing fallbacks for older browsers. Suppose we want to use some of our new viewport-related length units but ensure we have some kind of fallback for older browsers. Due to the way in which CSS is parsed, we can write the older value first, then the newer style one immediately afterward. Here's an example:

```
.thing {
    padding: 0 10vw;
    padding: 0 10dvw;
}
```

An older browser that understands `vw` but not `dvw` will use the first padding-based declaration and discard the second because it can't understand it. A newer browser that does support `dvw` units will use the first declaration and then immediately replace it with the second, as it comes after it.

Nice and simple. However, that is only really useful for minor fallbacks. There will be times when you need a more sophisticated and robust approach.

Feature queries

The native solution to forking code in CSS is to use **feature queries**, part of the CSS Conditional Rules Module Level 3 (`https://www.w3.org/TR/css3-conditional/`). Support was introduced in iOS and Safari 9, Firefox 22, Edge 12, and Chrome 28.

The great news is that feature queries follow a similar syntax to media queries. So, as you understand media queries, the way we use feature queries should seem instantly familiar. Consider this:

```
@supports (flashing-sausages: lincolnshire) {
    body {
            sausage-sound: sizzling;
            sausage-color: slightly-burnt;
            background-color: brown;
        }
    }
```

Here, the styles will only get applied if the browser supports the `flashing-sausages` property in combination with the `lincolnshire` value. I'm quite confident that no browser is ever going to support a `flashing-sausages: lincolnshire` property/value combination (and if they do, I want full credit), so none of the styles inside the `@supports` block will be applied.

Let's consider a more practical example. How about we use the `lch()` color function, which we cover in depth in *Chapter 8*, *CSS Color*, when browsers support it and fall back to a more limited color, to support, say, an older legacy browser, when they don't? Consider this example:

```
@supports (color: lch(50% 132 43)) {
    .Item {
        background-color: lch(50% 132 43);
    }
}
@supports not (color: lch(50% 132 43)) {
    .Item {
        background-color: red;
    }
}
```

Here, we are defining one block of code for when the browser supports a feature and another for when it doesn't. This pattern is fine if the browser supports `@supports` (yes, I realize that is confusing), but if it doesn't, it won't apply any of those styles.

If you want to make provision for devices that don't support `@supports` (and I have to tell you that would be very edge case at this point, as `@supports` is very well supported), you're better off writing your default declarations first and then your `@supports`-specific ones after. That way, the prior rule will be overruled if support for `@supports` exists, and the `@supports` block will be ignored if the browser doesn't support it. Our prior example could, therefore, be reworked to this:

```
.Item {
    background-color: red;
}
@supports (color: lch(50% 132 43)) {
    .Item {
        background-color: lch(50% 132 43);
    }
}
```

Sorry, that explanation was tough—hopefully you got through it! Notice also that you don't need to include the exact declaration that you will use in the rule. I used color as the "test" in the function, but used `background-color` in the rule. I therefore occasionally use `@supports` with a "test" that is a proxy for something else.

Using @supports as a proxy for a feature you can't test for

There is a new feature called `@starting-style` that allows you to define how you want elements to be styled when they are entering the DOM. We cover it in detail in *Chapter 12*, *Transitions, Transformations, and Animations*. All you need to know, for now, is that you can't use `@supports` to check for that `@starting-style`. But what we can do is use another, different property value pair that is a good proxy for the level of support we need. In this case, I know that `@starting-style` came out in Safari 17.5, and at the same time, `text-wrap-style` was introduced in Safari 17.5, so although using a different declaration in `@supports`, we can ensure that our styles only get applied when relevant. In this instance, you could nest a `@supports` block inside the selector like this.

```
@supports(text-wrap-style: auto) {
    opacity: 1;
    transform: none;
    transition: all .5s ease-in-out;
    @starting-style {
        opacity: 0;
        transform: translateX(-20px);
    }
}
```

Combining conditionals

You can also combine conditionals. Let's suppose we only wanted to apply some rules if both Flexbox and `pointer: coarse` were supported (in case you missed it, we covered the pointer interaction media feature back in *Chapter 3*, *Media and Container Queries*). Here is what that might look like:

```
@supports ((display: flex) and (pointer: coarse)) {
    .Item {
        display: inline-flex;
    }
}
```

Here, we have used the `and` keyword, but we can also use `or` alongside it in a query, or just instead of it; for example, if we were happy to apply styles if those two prior property/value combinations were supported, or if 3D transforms were supported:

```
@supports ((display: flex) and (pointer: coarse)) or
    (transform: translate3d(0, 0, 0)) {
    .Item {
        display: inline-flex;
    }
}
```

Note the extra set of parentheses that separates the `display` and `pointer` conditional from the `transform` conditional.

Finally, with all the newer selectors we have looked at and future ones coming in time, it is worth knowing you can test for those too. This is what a feature query for `:where()` would look like:

```
@supports selector(:where(div)) {
    /* styles */
}
```

You place the selector you are interested in testing support for into the `selector()` function.

All the related `not`-, `or`-, and `and`-related keywords work just the same with `selector()`. This is incredibly handy, as not only can we then fork our code on a specific feature, but we can do it for the more recent selectors introduced to CSS too.

The only other thing to mention is that you can nest other at-rules inside, such as media queries and keyframes.

That's everything you are likely to need to understand when it comes to feature queries. As with media queries, put your "default" styles first, and then your enhancement inside a feature query `@supports` at-rule.

Summary

At the beginning of this chapter, we considered how we can write our styles differently using CSS nesting, as a helpful “syntactic sugar” to visually encapsulate rules. We also considered CSS Layers, and where they are applicable, and the problems they can help with.

After that, we took a deep dive into how to select almost anything we need on the page with CSS’s powerful selection capabilities.

We also considered the newer selection technique of using `@scope` to limit selections by their position in the DOM.

Alongside all that, we also found time to look at the `@supports` at-rule for providing feature forks in our code.

Despite all the great new features and techniques that we covered in this chapter, we’ve still only touched the surface of what we can do with CSS.

In the next chapter, we are going to take a look at typography on the web. If you don’t know your `@font-face` from your `registered-axis`, come with me to *Chapter 7*, *Web Typography*, and all will be revealed.

7

Web Typography

Typography on the web has come on tremendously in the last few years. It was once necessary to limit ourselves to system fonts. Then, when custom fonts were made viable, a bevy of different file types was required to satisfy all the competing custom font implementations. Mercifully, things have now pretty much settled on `.woff` and the newer `.woff2` formats. In addition, we have more control over the particulars of our fonts in CSS, and now new variable fonts, which we will cover shortly, are popping up on a monthly basis.

If some of that introduction was lost on you, fear not. Let's try to get through the current state of the art in web typography. In this chapter, we will cover the following:

- System fonts for the fastest rendering
- Using custom fonts with online services
- Using `@font-face` to deliver custom web fonts
- Optimizing font loading
- Text alignment with `text-box`
- Controlling wrapping with `text-wrap`
- Variable fonts

First, however, let's talk about system fonts, as they represent the best-performing web typography choice you can opt for.

System fonts

Each operating system has its own set of fonts that come pre-installed. However, aside from a few exceptions, there aren't many fonts you can rely upon to be installed on every device a user might view your site with. Subsequently, we've grown accustomed to writing font **stacks**, which enable us to write a font "wish list" for the browser, as in this example:

```
font-family: -apple-system, BlinkMacSystemFont, Roboto, Ubuntu, "Segoe UI", "Helvetica Neue", Arial, sans-serif;
```

The browser reads this declaration and goes from left to right until it finds a font it has available on the system, choosing that to render the associated text. In our example system font stack here, macOS users will get San Francisco or Helvetica (`-apple-system` tells Safari to pick San Francisco, and `BlinkMacSystemFont` tells Chrome to use Helvetica). Android doesn't have those fonts, so it will continue along the stack until it gets to `Roboto`, which it does have, and render the text with that. Meanwhile, the popular Ubuntu Linux distribution will likely get to `Ubuntu`, while Windows users will likely not get a match until `Segoe UI`, and then, in our stack, we add some last-ditch attempts to get something nice with `Helvetica Neue` or `Arial`. Then, if all else fails, we tell the browser to use any `sans-serif` font it has. One way or another, a font will be selected to display the text without requesting an external file.

> Some font names contain whitespace, so it is necessary to surround those strings with single or double quotes. You can see in our previous example that both `"Segoe UI"` and `"Helvetica Neue"` are written this way.

For a lot of situations, using system fonts is a compelling choice. There is zero network overhead, and you'll never have to worry about fonts not loading or seeing unsightly jumps on the page as one font is replaced with another. Although, as we shall learn shortly, there are means to mitigate that.

The trade-off is that with system fonts, you can never be entirely sure how your text is being rendered on the user's device. Your page might be rendered with Roboto for some, and San Francisco for others. If specific typography is crucial to the project at hand, you'll want to look at web fonts, which use `@font-face` to load a specific font.

The @font-face CSS rule

The `@font-face` CSS rule has been around since CSS 2 (but was subsequently absent in CSS 2.1). It was even supported partially by Internet Explorer 4 (no, really)! So, what's it doing here, when we're supposed to be talking about the latest CSS?

Well, as it turns out, `@font-face` was reintroduced for the CSS Fonts module (`https://www.w3.org/TR/css3-fonts`). Due to the historic legal quagmire of using fonts on the web, it took years to gain traction as the *de facto* solution for web typography.

Like anything on the web that involves assets, in the beginning, there was no single file format for font delivery. The **Embedded OpenType** (files with an `.eot` extension) font was Internet Explorer's, and not anyone else's, preferred choice. Others favored the more common **TrueType** (the `.ttf` file extension), while there was also **Scalable Vector Graphics** (`.svg`), and then the **Web Open Font Format** (the `.woff`/`.woff2` extension). Thankfully, nowadays you really only need to consider WOFF, although there are two types: WOFF and WOFF2. Where a browser supports WOFF2, that will always be preferable, as it is simply a more efficient way of compressing the font information.

Using custom fonts with online services

There are a number of online font services for getting beautiful typefaces onto the web. Google Fonts and Adobe Fonts are probably the two most popular. They each have their own variation of the required syntax for getting their fonts onto your websites. Essentially, they provide a section of code to stick into the `<head>` of your HTML. Here is an example font, Open Sans, from Google Fonts:

```
<link rel="preconnect" href="https://fonts.googleapis.com">
<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
<link href="https://fonts.googleapis.com/css2?family=Open+Sans&display=swap"
    rel="stylesheet">
```

And then, you use the font name provided as the value for your font-family property. This is by far the simplest way of adding a bespoke font to your page, plus you know all the legalities of using the font are addressed. It also does a good job of abstracting away some of the implementation details.

However, it's not always possible or preferable to use an online font provider. You may be stuck behind a corporate firewall (or state firewall!) that doesn't allow a certain provider, or you may not want the financial overhead or service availability of a third-party provider. Whatever the reason may be, it is worth knowing how to do it for ourselves.

Implementing web fonts with @font-face

For this exercise, I'm going to use "Inter" by Rasmus Andersson. It's the font used for the majority of the `https://rwd.education` website. Not only that, it's also what's known as a **variable** font.

Before we look at variable fonts, let's look at the kind of web fonts that you will likely spend the majority of your time dealing with. If you grab the Inter font from `https://rsms.me/inter/`, you will have both the standard and variable fonts at your disposal.

If you can, download a **subset** of your font, specific to the language you intend to use. For example, if you are not going to be using Cyrillic or Greek characters, why include them? The resultant file size will be much smaller as it won't contain glyphs for languages you have no intention of using. You can often choose character sets when you are buying a font, but for an open source version, there are a growing number of services and utilities, such as **Glyphhanger** (`https://github.com/zachleat/glyphhanger`), that can subset a font for you.

Having downloaded the Inter font, a look inside the ZIP file reveals folders of the different files available. I'm choosing the `Inter-Regular.woff2` and `Inter-Regular.woff` files for our example.

To make a web font available for use in our CSS, we use the `@font-face` at rule. This lets us provide a name for the font that we can then reference, and also tells the browser where to go to fetch this file. Let's look at the syntax:

```css
@font-face {
    font-family: "InterRegular";
    src: url("Inter-Regular.woff2") format("woff2"),
        url("Inter-Regular.woff") format("woff");
    font-weight: normal;
    font-style: normal;
    font-display: fallback;
}
```

Inside the `@font-face` braces, we give our font the `"InterRegular"` name. I could have called it `"FlyingBanana"` had I wanted; it wouldn't affect anything other than I would then need to reference this font with that name when I wanted to use it. Here is an example:

```css
.hero-Image {
    font-family: "FlyingBanana", sans-serif;
}
```

So, know that you can do this, but it makes sense to use a name similar to the font you are expecting to see!

You can then see two successive `url()` and `format()` sections within our `src` property. Each pair of these is separated by a comma. As we noted previously, `.woff2` is always our preference, as it is a better-compressed version of the same font, so we list that first. As long as the browser can understand `.woff2` and download the file, it will do so. If, however, the browser can't find that file or can't understand `.woff2`, it will download the `.woff` file listed afterward.

Now, although this block of code is great for fans of copy and paste, it's important to pay attention to the paths the fonts are stored in. For example, if we were to place fonts in a folder inventively called `fonts` on the same level as a `css` folder, we would need to amend our paths. So, our `@font-face` block would become the following:

```
@font-face {
    font-family: "InterRegular";
    src: url("../fonts/Inter-Regular.woff2") format("woff2"),
        url("../fonts/Inter-Regular.woff") format("woff");
    font-weight: normal;
    font-style: normal;
    font-display: fallback;
}
```

With the correct path, the font will display. Most of the time, when your font isn't displaying, or displays fine on your local system but not online, it's due to an error in the URL used to reference the font.

> You can see in this example that a `font-weight` value of `normal` is used. In most circumstances, I prefer to use the equivalent numeric values for `font-weight`, such as `400` for normal weight, `600` for semi-bold, `700` for bold, and so on. You should know that you can also use relative terms such as `lighter` and `bolder`, which work in relation to their parent, but I have always found using those too unpredictable in practice for my needs.

Optimizing font loading with font-display

If your main font is a web font, it's a good idea to request the file upfront by loading it with a link in the `head` section of your HTML with the `rel` attribute value as `preload`, as in this example:

```
<link
    rel="preload"
    href="fonts/inter.var.woff2"
    as="font"
    type="font/woff2"
    crossorigin
/>
```

> You can read more about this font optimization technique here: `https://web.dev/fast/#prioritize-resources`.

Adding a link with the `rel="preload"` attribute and value triggers a request for the web font early in the critical rendering path, without having to wait for the **CSS Object Model** (**CSSOM**) to be created. More simply, it makes browsers such as Chrome go and fetch the font immediately. While this technique is recommended by Google, even if other browsers don't do the same, it is unlikely to do any harm. Note, if you are serving up both a `.woff` and `.woff2` file, it's only worth doing this optimization for the `.woff2` file; only browsers that support `.woff2` also support the `preload` value in the `rel` attribute, and all the evergreen browsers already support WOFF2.

We can make further optimizations with web fonts by making use of the `font-display` property.

font-display

We can also make use of the `font-display` property of CSS (older browsers that don't understand it will simply ignore it):

```
font-display: fallback;
```

You can also see this property being used in the previous examples. It provides some control over how fonts should be displayed.

The `fallback` value in our example code sets an extremely short `block` period and a short `swap` period.

To understand what terms such as **block** and **swap** mean in this context, we need to consider what the browser does in terms of displaying fonts. I'm describing things generically here, but the concepts work for our needs.

Imagine a browser loading our web page in which we have specified a web font for our text. The browser already has the HTML and is parsing the CSS, and learns that it needs to download a font in order to display the text it has as intended. Before it draws any text to the screen, it hangs on, waiting for the web font so it can paint the text onto the page as needed. This delay is referred to as a **Flash Of Invisible Text** (**FOIT**).

As soon as the font arrives, the browser parses it and paints the text to the screen accordingly.

The hope is that this delay is imperceptible. Where it is not, there are two schools of thought on how best to handle things.

One option is to wait for the font to be downloaded, usually for up to a few seconds but sometimes indefinitely; Safari is the most famous proponent of this option.

The second option is to render the text with a system font initially and then replace the font with the correct font when the browser has it.

This redraw of text from system font to the actual intended font is known as a **Flash of Unstyled Text (FOUT)**.

All the `font-display` setting does is allow us some control over what we would like to see happen.

The possible values are as follows:

- `auto`: This will be whatever the browser determines is most appropriate.
- `block`: Get a white screen for up to 3 seconds (but the delay is ultimately at the browser's discretion), and then the actual font can replace any system displayed font at any future point.
- `swap`: There is a very short blocking period (100 ms is the recommended amount) to let the web font load; otherwise, a system font shows, and the web font can be swapped in whenever it is available.
- `fallback`: This option prevents a web font from replacing a system font if a set amount of time has passed (3 seconds is the recommendation). This option blocks for around 100 ms initially and allows a swap for up to 3 seconds, but after that, if the web font subsequently arrives, it doesn't get applied.
- `optional`: Here, the browser allows a very short duration for the web font to load (100 ms) but no swap period. The result of this is that the browser has the option of canceling the font download if it hasn't arrived, or if it has, using it for subsequent page loads.

You can read the specification on this property here: `https://www.w3.org/TR/css-fonts-4/#descdef-font-face-font-display`.

There are plenty of other font-related properties and values specified in the CSS Font Module Level 4 specification, but `font-display` is currently the most widely implemented and has the most direct relevance to responsive web designs and performance.

Before we get to variable fonts, there are a few more recently added niceties to web typography I want to tell you about.

Consolidating the appearance of two fonts with font-size-adjust and size-adjust

Because different fonts have different size characteristics, two different fonts, given the same `font-size` value, can appear to be different sizes. This can be jarring when one font is replaced with another, such as when a system font is displayed and is then replaced by a web font once the resource has loaded from the network. It would be preferable to ensure the sizes, even if not the style, of the fonts were consistent between the two. Or perhaps we want to ensure that when a different font is used inside a paragraph of another font, the sizing is consistent, given that they share the same `font-size` value.

We can address this problem with the `font-size-adjust` property, which is kind of like a marsupial, in that it also includes the `size-adjust` property. Both these properties do something to fonts, but not the same thing. Cripes, this is already confusing. Let's try and make sense of each.

The font-size-adjust property

This property, which can be applied to any piece of text, allows us to change the apparent size of text, typically useful when font fallback occurs, but also when a font is used inside some other flowing text, such as a code sample inside normal article copy.

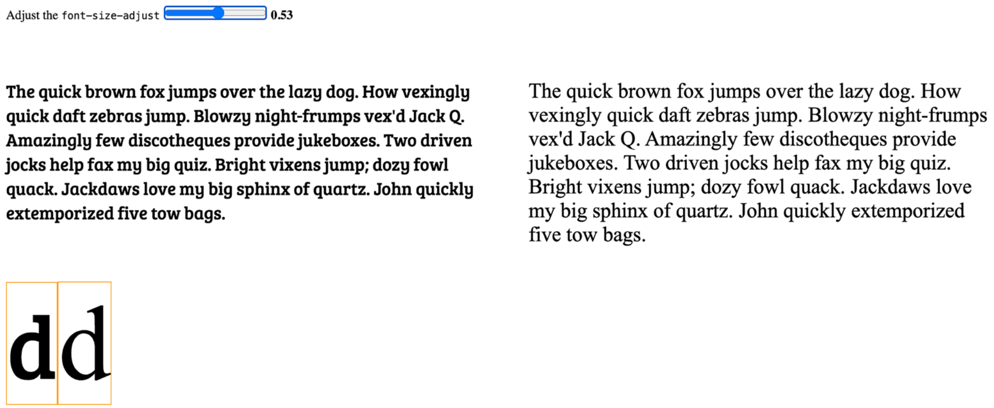
To best understand what this property does, open `example_07-01` in the browser.

We have two paragraphs of text made up of pangrams. Each paragraph is set with a different font. The first is the Bree web font, loaded in from Google Fonts, and the second is the more ubiquitous Times New Roman, which can be found bundled on many operating systems. Incidentally, both those paragraphs are set to be editable, so you may wish to paste some other text in for comparison.

Below the two paragraphs, you can see a larger sample glyph from each font, with a border around it to display its bounds.

The goal with `font-size-adjust` is to make the two fonts display at the same size, given the same `font-size` value, so that if one were replaced with the other, the content would remain equally legible.

At the top of the page, there is a range input that lets you adjust the default `font-size-adjust` value from `0.5`. See what happens when you adjust that slider.



Adjust the `font-size-adjust` 0.53

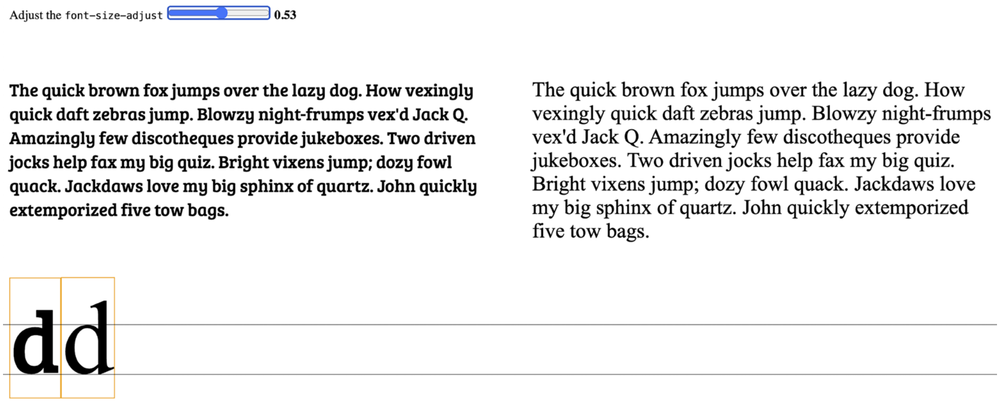
The quick brown fox jumps over the lazy dog. How vexingly quick daft zebras jump. Blowzy night-frumps vex'd Jack Q. Amazingly few discotheques provide jukeboxes. Two driven jocks help fax my big quiz. Bright vixens jump; dozy fowl quack. Jackdaws love my big sphinx of quartz. John quickly extemporized five tow bags.

The quick brown fox jumps over the lazy dog. How vexingly quick daft zebras jump. Blowzy night-frumps vex'd Jack Q. Amazingly few discotheques provide jukeboxes. Two driven jocks help fax my big quiz. Bright vixens jump; dozy fowl quack. Jackdaws love my big sphinx of quartz. John quickly extemporized five tow bags.

Figure 7.1: Use the slider in the example to adjust the value and match the display of the fonts

The specification provides an example at `https://www.w3.org/TR/css-fonts-5/#example-771aee48` explaining the following: *Starting with a value of 0.5, the aspect value can be adjusted until the borders around the two letters line up.*

However, in practice, I have found that this isn't always true. In our example, lining the two boxes up does not produce the best match between the paragraphs. I've added horizontal lines here on the image to show that the x heights on those two glyphs are different when the bounding boxes are the same height.



Adjust the `font-size-adjust` 0.53

The quick brown fox jumps over the lazy dog. How vexingly quick daft zebras jump. Blowzy night-frumps vex'd Jack Q. Amazingly few discotheques provide jukeboxes. Two driven jocks help fax my big quiz. Bright vixens jump; dozy fowl quack. Jackdaws love my big sphinx of quartz. John quickly extemporized five tow bags.

The quick brown fox jumps over the lazy dog. How vexingly quick daft zebras jump. Blowzy night-frumps vex'd Jack Q. Amazingly few discotheques provide jukeboxes. Two driven jocks help fax my big quiz. Bright vixens jump; dozy fowl quack. Jackdaws love my big sphinx of quartz. John quickly extemporized five tow bags.

Figure 7.2: With matching bounding boxes, you don't always get the best visual appearance of a paragraph

By default, `font-size-adjust` works on the x height of the font (the size of a lowercase x glyph in the font), but you can also use a different metric to adjust, such as the height of uppercase letters (`cap-height`). To use something other than the default, use two values: first, the metric to be adjusted against, and then the value. Here is an example:

```
font-size-adjust: cap-height 0.6;
```

You can also use `from-font` as the value, which uses that metric from the first font it can use in the relevant font stack applied:

```
font-size-adjust: ex-height from-font
```

However, I have had mixed results when using `from-font` so I tend to rely on "eye-balling" the value based on seeing the two fonts side-by-side, as in our example.

You can investigate the other, more esoteric possibilities in the specification here: `https://www.w3.org/TR/css-fonts-5/#font-size-adjust-prop`.

The size-adjust property

The `size-adjust` property only works on `@font-face` rules. It is used as a multiplier for the font.

Much like `font-size-adjust`, which we just looked at, `size-adjust` is useful when trying to harmonize the size of fonts. Sadly, when I tested it, it does not allow updating with a dynamic value; you have to set the value in your `@font-face` rule, and reload the font each time to see the changes:

```
@font-face {
    font-family: 'Bree Serif';
    font-style: normal;
    font-weight: 400;
    font-display: swap;
    src: url(https://fonts.gstatic.com/s/breeserif/
v17/4UaHrEJCrhhnVA3DgluA96rp57F2IwM.woff2) format('woff2');
    size-adjust: 90%;
}
```

Values greater than `100%` adjust the size of the rendered font for a given `font-size` up, and values less than `100%` adjust the size down.

You can read the specification for `size-adjust` here: `https://www.w3.org/TR/css-fonts-5/#size-adjust-desc`.

Better text alignment with text-box

Let's start with a fairly common problem. Consider this button:

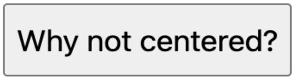


Figure 7.3: This might appear fine, or you might be feeling uncomfortable

We have straightforward markup:

```
<button class="btn" type="button">Why not centered?</button>
```

We also have minimal CSS:

```
.btn {
    height: 45px;
    font-size: 18px;
    font-family: -apple-system, BlinkMacSystemFont, Roboto, Ubuntu,
        "Segoe UI", "Helvetica Neue", Arial, sans-serif;
}
```

I'm applying the system font stack we looked at earlier in the chapter to the button, as buttons get their own default font stack, which I rarely want. Now, many of you will look at that button and ask, What's the problem? Most of the time, I'd be in that same group of people. However, I know full well there are some of you who are already feeling uncomfortable. The thing is, there is a little more space above the letters than below. And that may be a problem.

Different fonts have glyphs of different heights, as they go from the very top of the highest character to the bottom of the very bottom of the lowest ascender; the part of a character that dangles below the baseline, such as the curly bit of a letter "g."

Historically, when a designer has asked for that issue to be fixed, we have needed to artificially add a little padding above or below the text to get it visually centered, but this is a sub-optimal solution. Another font might not have the same space around each glyph, and so you may fix for one presentation in your font stack, but make another worse.

The `text-box-edge` and `text-box-trim` properties, and their `text-box` shorthand, are designed to help just such a scenario. They allow us to remove space from around the characters. They are new properties, still not in Firefox as of mid-2025, but usable in all other evergreen browsers.

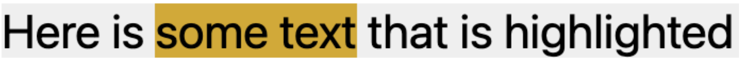
To control how much space is cut from the edge of the font glyphs, we use `text-box-edge`. To control which edge we are cutting from, such as top/over or bottom/under, we use `text-box-trim`.

To help illustrate, open `example_07-02`. After the button we already considered, you can see some text with a colored background applied, and a section of text with a golden background. This has simple markup:

```
<p>Here is <span>some text</span> that is highlighted</p>
```

We are going to play with the properties, which will alter the background color by effectively trimming away the unneeded space around the glyphs.

Here is how things look by default with no settings applied:



Here is some text that is highlighted

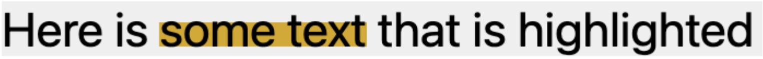
Figure 7.4: The default spacing around glyphs showing

As stated, the `text-box-trim` property is used to define where the excess should be cut away. For Western languages, the space above glyphs is the block start, or over edge. The space below is the block end, or under edge. The values we enter into `text-box-edge`, or the relevant position in the `text-box` shorthand, control where to cut, such as above and below. Let's add some values to our example just to see an effect. I'm going to amend the CSS on that span to this:

```
 span {
    background-color: goldenrod;
    text-box: trim-both ex alphabetic;
}
```

If you do the same and refresh the page, you will see that nothing has happened. What?

The first important point is that `text-box` works on block containers of text and inline boxes. `span` is an inline element by default, hence nothing happens. Let's change that `span` element to have a display of `inline-block` and refresh:



Here is some text that is highlighted

Figure 7.5: The text-box properties only work on block containers

That's better. Now, you can see that the background color around the span is much shorter. But what do `trim-both`, `ex`, and `alphabetic` actually do? I've used the shorthand in our example, and the first value is for `text-box-trim`.

text-box-trim possible values

You can set it to the following:

- `trim-both`: This removes space from over and under the characters
- `trim-start`: This removes space from just the starting edge, which would be above the characters in a Western language
- `trim-end`: This removes space from the bottom, or the under edge, in a Western language

text-box-edge possible values

For `text-box-edge`, you can use one or two values. If just one is supplied, the edge applies to both over and under. If two, the first is for the over, and the second is for the under.

Possible values for the over edge are as follows:

- `text`: This is the default space
- `cap`: This is the top of the capital letters
- `ex`: This is the top of the short lowercase letters, such as "n" or "a"

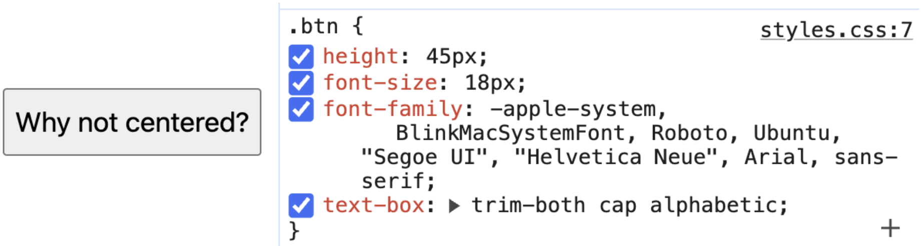
And possible values for the under edge are as follows:

- `text`: This is the default space
- `alphabetic`: This is the bottom of the short lowercase letters, such as "n" and "a"

If you play around with those values, you can see the different effects. For our own little problem, we can line the text in our button up perfectly by taking the space away at the top/over edge of the capital letters, and the space away from the under edge/bottom of the lowercase characters by adding this declaration:

```
text-box: trim-both cap alphabetic;
```

And just like that, the pixel pedants among you can breathe a sigh of relief as the text sits perfectly centered within the button:



```
.btn {                                          styles.css:7
☑ height: 45px;
☑ font-size: 18px;
☑ font-family: -apple-system,
      BlinkMacSystemFont, Roboto, Ubuntu,
    "Segoe UI", "Helvetica Neue", Arial, sans-
    serif;
☑ text-box: ▶ trim-both cap alphabetic;
}                                                 +
```

Figure 7.6: If you like your designs pixel perfect, text-box is a great tool

It is also worth appreciating that the value of `text-box` is not inherited, so block containers inside existing block containers do not automatically get the same `text-box` settings applied. Also, `text-box-edge` has no effect if a value for `text-box-trim` is not set.

Controlling the flow of text with text-wrap

The `text-wrap` property is actually a shorthand for the `text-wrap-mode` and `text-wrap-style` properties. These properties provide the ability to better control the way text wraps within its container. I have never found the need to split these properties out in practice, as even the shorthand only takes a single keyword, so I am not going to suggest you do either.

You should know up front that using some of the `text-wrap` property and value combinations can cause a performance hit. I don't think that is something you should be overly concerned about, though, for two reasons. Firstly, layout algorithms only ever get performed faster over time, as computing speed increases, and browser algorithms are made more efficient. Secondly, browsers limit how many lines of text the worst offending values can be applied to as a fail-safe anyway.

If you are following along with your text editor open, take a look at `example_07-03`. You will see the same pangram text repeated in the center column a number of times. All the text is the same size, in the same container, but with a different `text-wrap` combination applied:

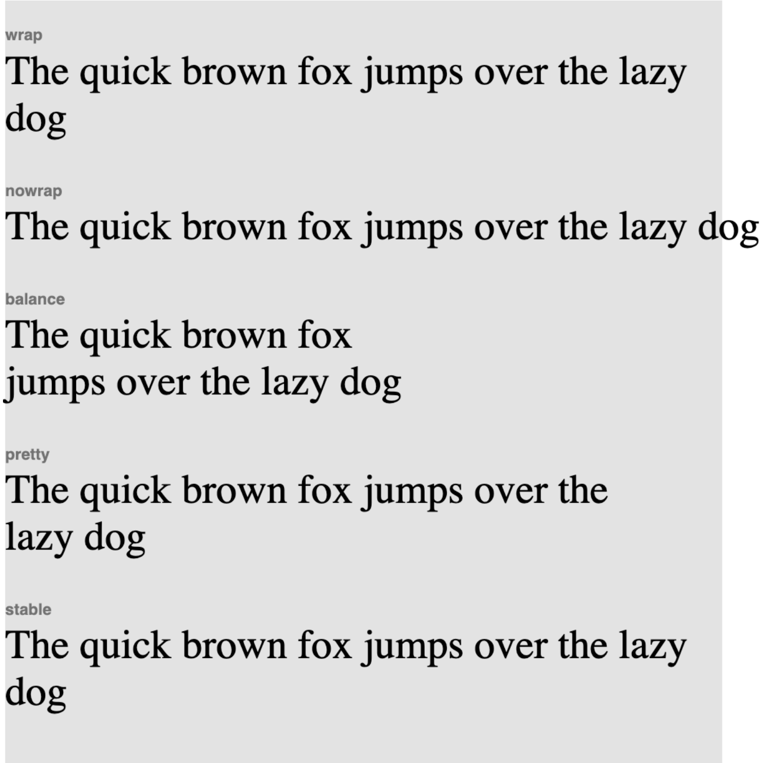

wrap
The quick brown fox jumps over the lazy
dog
nowrap
The quick brown fox jumps over the lazy dog
balance
The quick brown fox
jumps over the lazy dog
pretty
The quick brown fox jumps over the
lazy dog
stable
The quick brown fox jumps over the lazy
dog

Figure 7.7: Same text with different text-wrap values applied to change the wrapping

So, we can see the differences, but what are the different values actually doing?

- `wrap`: This is the default manner in which text gets laid out.
- `nowrap`: This stops the text wrapping in the container. For almost all practical purposes, it is the same as setting `white-space: nowrap`.
- `balance`: The browser looks at the characters and decides the most balanced way of wrapping the text.
- `pretty`: This primarily prevents orphan words at the end of paragraphs. This one is good for body copy in articles and the like.
- `stable`: This is only useful if the element in question is an editable area. It ensures that lines before the one being edited don't jump around due to re-wrapping.

You should be aware that the `text-wrap` property and value combinations only work on the spaces between words, known as **soft breaks**. If you have a `<br/>` break return element in there, as shown on the example page, you will still get a hard return at that point.

In broad strokes, when I use `text-wrap`, I tend to apply `balance` on article headings, and `pretty` on body copy, but there is no right answer here; you must use your own judgment on the correct aesthetics for your project.

So far, we have looked at how to get font files into our project and even considered how best to deal with them from a performance point of view.

However, the most recent development in web fonts, and probably the one that has most developers excited, is **variable** fonts. “What new witchery is this?” I hear you cry. Let’s find out.

Variable fonts

A “normal” font contains the information and glyphs for one variation of a typeface: the regular version of Roboto, for example. By comparison, a variable font in a single file would contain everything needed for every variation of Roboto. Bold, italic, thin, black, medium, and more besides!

This new devilry is not without consequence. A variable version of a font is usually considerably larger in file size terms than a “normal” version. However, it can still make sense when you are making heavy use of a single typeface. It also makes an almost limitless variation of a font possible, effectively making a custom version of a font possible.

Caveats aside, let’s look at what we can do with a variable font.

font-face changes

I’m working with the Inter font we used before, but using the variable version instead. First, let’s consider how we tell the browser we are working with a variable font in the first place. It’s the `@font-face` syntax again, but with a few changes:

```
@font-face {
    font-family: "Inter-V";
    src: url("fonts/inter.var.woff2") format("woff2-variations");
    font-weight: 100 900;
    font-style: oblique 0deg 10deg;
    font-display: fallback;
}
```

The first thing to note is `format`. We are setting this to `woff2-variations` to tell the browser this is a font file that makes use of variations.

The next thing to keep in mind is that we are using a range for `font-weight`. The two-word value syntax we are using here is only understood by browsers that understand variable fonts. It is a way of telling the browser the range of weights this font can use. While `font-weight` has a range limit of 0–999, other properties may have a different range. Others still simply accept a binary value of `1` or `0`.

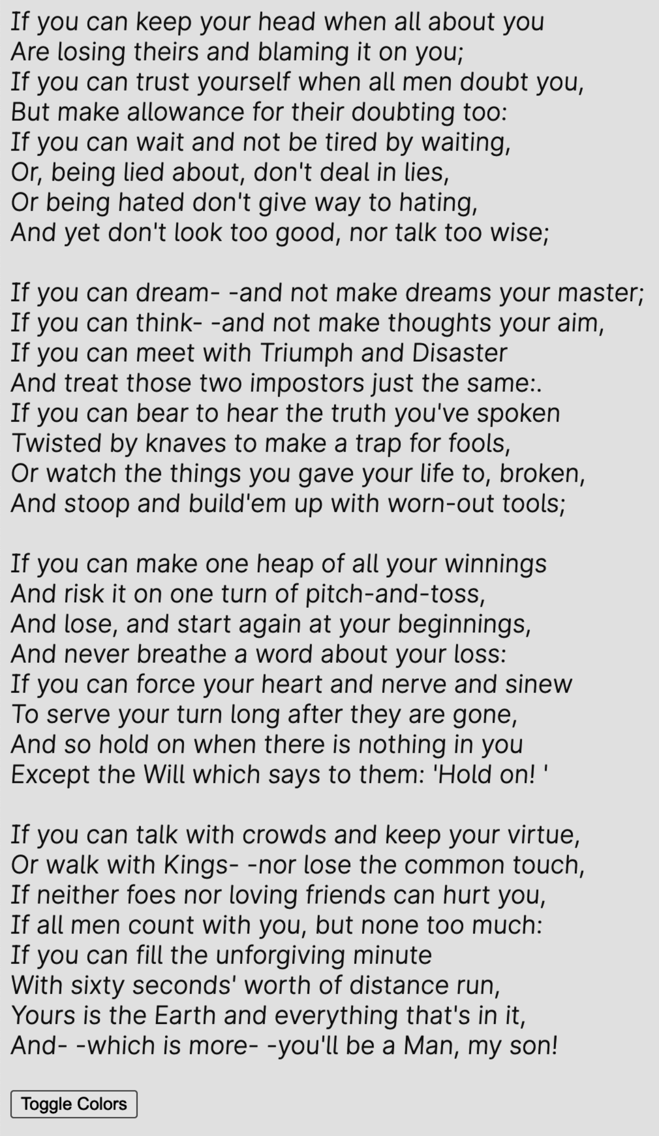
Another thing to remember is that we have `font-style`, which is also being provided with a multiple-value syntax. The `oblique` keyword tells the browser the next values relate to how oblique the font can be, and the two values after that define the range. Be aware that although the range here is positive, you can pass in negative values. We will see that in use in a moment.

Finally, note that we have used the `font-display` property here to tell the browser how we want to handle the loading and display of this font, but that is no different than non-variable fonts.

So that's setting up the variable font, but then there is how to actually apply it. Take a look at this code:

```
@supports (font-variation-settings: "wdth" 200) {
    body {
        font-family: "Inter-V";
        font-variation-settings: "wght" 300, "slnt" -4;
        font-feature-settings: "cv08", "cv05";
    }
}
```

You can see what text looks like with these settings by opening `example_07-04`, which is the poem **If**, by Rudyard Kipling:



If you can keep your head when all about you
Are losing theirs and blaming it on you;
If you can trust yourself when all men doubt you,
But make allowance for their doubting too:
If you can wait and not be tired by waiting,
Or, being lied about, don't deal in lies,
Or being hated don't give way to hating,
And yet don't look too good, nor talk too wise;

If you can dream- -and not make dreams your master;
If you can think- -and not make thoughts your aim,
If you can meet with Triumph and Disaster
And treat those two impostors just the same:.
If you can bear to hear the truth you've spoken
Twisted by knaves to make a trap for fools,
Or watch the things you gave your life to, broken,
And stoop and build'em up with worn-out tools;

If you can make one heap of all your winnings
And risk it on one turn of pitch-and-toss,
And lose, and start again at your beginnings,
And never breathe a word about your loss:
If you can force your heart and nerve and sinew
To serve your turn long after they are gone,
And so hold on when there is nothing in you
Except the Will which says to them: 'Hold on! '

If you can talk with crowds and keep your virtue,
Or walk with Kings- -nor lose the common touch,
If neither foes nor loving friends can hurt you,
If all men count with you, but none too much:
If you can fill the unforgiving minute
With sixty seconds' worth of distance run,
Yours is the Earth and everything that's in it,
And- -which is more- -you'll be a Man, my son!

Figure 7.8: Text set with a variable font

So, let's dig into these new settings and understand how to make use of them.

How to make use of a variable font

Variable fonts use what's termed a **variation axis**.

A variation axis is just a way of defining two points at either end of a scale. You can then choose any point along the scale to display your font. The scale doesn't need to be vast; in fact, it can be as simple as "on" or "off" (not much of a scale, I know). Variation axes are subdivided into two groups: registered and custom.

Registered axis

Registered axes are the most popular ones, for which the specification writers have made specific provision by providing keywords and/or values to give us easy access:

- **Weight**: How heavy the text appears, for example, `font-weight: 200`.
- **Width**: How narrow (condensed) or wide the text appears, for example, `font-stretch: 110%`.
- **Italic**: Whether the font is being displayed as italic or not, for example, `font-style: italic`.
- **Slant**: Don't confuse this with `italic`. This simply alters the angle of the text; it doesn't substitute any glyphs. An example is `font-style: oblique 4deg`.
- **Optical size**: This is the only one of the registered axes that has required a new font property. Using `font-optical-sizing` lets you alter, yes, you guessed it, the optical sizing. But what is that? It's the practice of altering a glyph based on the size it is displayed to aid clarity. This means the same glyph displayed at a large size might enjoy thinner stems, for example.

The values you choose to use with these properties should fall within the capabilities of the variable font you are using. For example, there isn't much point in specifying a font weight of `999` if the font you are using can only go up to `600`.

There is also a low-level property that lets you bundle up your variable font settings into a single property/value combination, which is what we have done in our example:

```
font-variation-settings: "wght" 300, "slnt" -4;
```

Here, we have set the font weight to `300` and the angle of the slant to `-4`. However, the specification offers the following cautionary advice:

When possible, authors should generally use the other properties related to font variations (such as font-optical-sizing), and only use this property for special cases where its use is the only way of accessing a particular infrequently used font variation. For example, it is preferable to use font-weight: 700 rather than font-variation-settings: “wght” 700.

I’m not sure why this advice is given. The specification doesn’t explain the rationale. You can find the current version of the specification here: `https://www.w3.org/TR/css-fonts-4/`.

> Font variation properties can be animated and transitioned, which can make for some fantastic effects!

Having covered the registered axis group, let’s briefly take a look at the custom axis group.

Custom axis

Variable fonts can include their *own* axes. For example, the FS Pimlico Glow VF font has a “glow” axis. You can amend that like this:

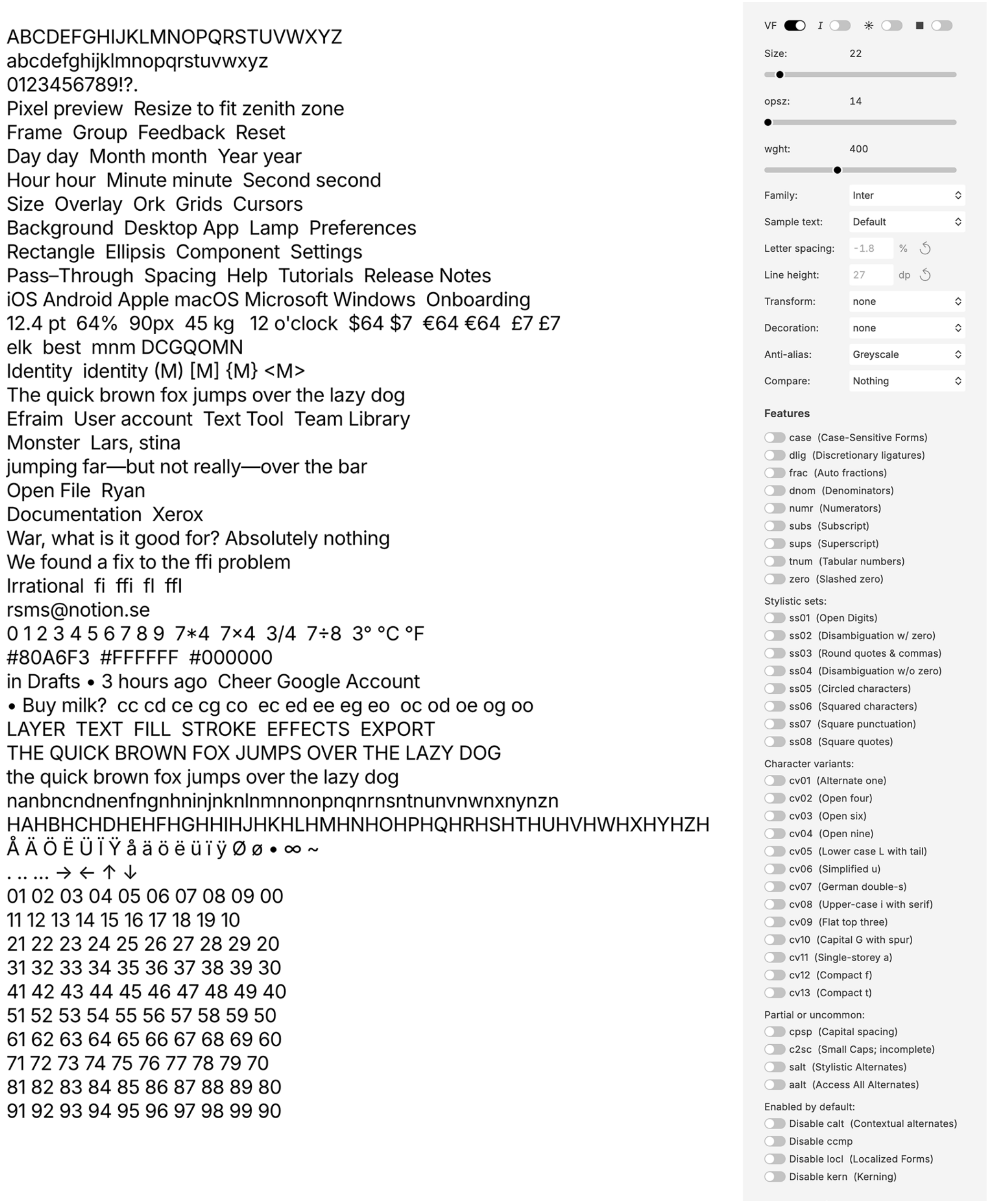
```
font-variation-settings: "GLOW" 500;
```

Notice how this custom axis is written in uppercase, that is, `"GLOW"`? That’s how to determine the difference between a registered and custom axis in variable font settings.

If your head isn’t already hurting from the seemingly endless possibilities, I also need to tell you about font features, which are sort of similar to the two different axes we just looked at. Don’t worry; it will make sense shortly.

Font features

Variable fonts can also include their own **features**. These features can be literally anything the font designer decides to dream up! Take a look at the choices for Inter:



ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789!?.
Pixel preview Resize to fit zenith zone
Frame Group Feedback Reset
Day day Month month Year year
Hour hour Minute minute Second second
Size Overlay Ork Grids Cursors
Background Desktop App Lamp Preferences
Rectangle Ellipsis Component Settings
Pass–Through Spacing Help Tutorials Release Notes
iOS Android Apple macOS Microsoft Windows Onboarding
12.4 pt 64% 90px 45 kg   12 o'clock $64 $7 €64 €64 £7 £7
elk best mnm DCGQOMN
Identity identity (M) [M] {M} <M>
The quick brown fox jumps over the lazy dog
Efraim User account Text Tool Team Library
Monster Lars, stina
jumping far—but not really—over the bar
Open File Ryan
Documentation Xerox
War, what is it good for? Absolutely nothing
We found a fix to the ffi problem
Irrational fi ffi fl ffl
rsms@notion.se
0 1 2 3 4 5 6 7 8 9 7*4 7×4 3/4 7÷8 3° °C °F
#80A6F3 #FFFFFF #000000
in Drafts • 3 hours ago Cheer Google Account
• Buy milk? cc cd ce cg co ec ed ee eg eo oc od oe og oo
LAYER TEXT FILL STROKE EFFECTS EXPORT
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
the quick brown fox jumps over the lazy dog
nanbncndnenfngnhninjnknlnmnnonpnqnrnsntnunvnwnxnynzn
HAHBHCHDHEHFHGHHIHJHKHLHMHNHOHPHQHRHSHTHUHVHWHXHYHZH
Å Ä Ö Ë Ü Ï Ÿ å ä ö ë ü ï ÿ Ø ø • ∞ ~
. .. ... → ← ↑ ↓
01 02 03 04 05 06 07 08 09 00
11 12 13 14 15 16 17 18 19 10
21 22 23 24 25 26 27 28 29 20
31 32 33 34 35 36 37 38 39 30
41 42 43 44 45 46 47 48 49 40
51 52 53 54 55 56 57 58 59 50
61 62 63 64 65 66 67 68 69 60
71 72 73 74 75 76 77 78 79 70
81 82 83 84 85 86 87 88 89 80
91 92 93 94 95 96 97 98 99 90

VF ● *I* ○ ☼ ○ ■ ○

Size: 22
opsz: 14
wght: 400
Family: Inter
Sample text: Default
Letter spacing: -1.8 %
Line height: 27 dp
Transform: none
Decoration: none
Anti-alias: Greyscale
Compare: Nothing

Features

○ case (Case-Sensitive Forms)
○ dlig (Discretionary ligatures)
○ frac (Auto fractions)
○ dnom (Denominators)
○ numr (Numerators)
○ subs (Subscript)
○ sups (Superscript)
○ tnum (Tabular numbers)
○ zero (Slashed zero)

Stylistic sets:
○ ss01 (Open Digits)
○ ss02 (Disambiguation w/ zero)
○ ss03 (Round quotes & commas)
○ ss04 (Disambiguation w/o zero)
○ ss05 (Circled characters)
○ ss06 (Squared characters)
○ ss07 (Square punctuation)
○ ss08 (Square quotes)

Character variants:
○ cv01 (Alternate one)
○ cv02 (Open four)
○ cv03 (Open six)
○ cv04 (Open nine)
○ cv05 (Lower case L with tail)
○ cv06 (Simplified u)
○ cv07 (German double-s)
○ cv08 (Upper-case i with serif)
○ cv09 (Flat top three)
○ cv10 (Capital G with spur)
○ cv11 (Single-storey a)
○ cv12 (Compact f)
○ cv13 (Compact t)

Partial or uncommon:
○ cpsp (Capital spacing)
○ c2sc (Small Caps; incomplete)
○ salt (Stylistic Alternates)
○ aalt (Access All Alternates)

Enabled by default:
○ Disable calt (Contextual alternates)
○ Disable ccmp
○ Disable locl (Localized Forms)
○ Disable kern (Kerning)

Figure 7.9: On the right of the Inter settings page are the enormous possibilities for the font

All of those "features" are settings that can be turned on with CSS. You can play around with these settings yourself here: `https://rsms.me/inter/lab/?varfont=1`.

When we want to apply these custom features, we make use of the `font-feature-settings` property. This works very similarly to the `font-variation-settings` syntax. For example, to switch on "slashed zeros," we can do this:

```
font-feature-settings: "zero";
```

This "slashed zeros" option is a binary choice, so it is not necessary to write `"zero" 1`. However, if we wanted to toggle this setting off further into our styles, we could write the following:

```
font-feature-settings: "zero" 0;
```

When you want to apply multiple `font-feature-settings`, you can comma-separate them. So, if we wanted to have "lowercase *L* with tail" and "uppercase *i* with serif," we could write this:

```
font-feature-settings: "cv08", "cv05";
```

> When looking at the chapter code, don't concern yourself with any additional syntax that doesn't make sense. There are CSS custom properties used, for example, and we will cover those fully in *Chapter 13*, *Custom Properties and CSS Functions*.

There is plenty more in the way of settings and features specified in CSS Fonts Module Level 4. We have just scratched the surface! To dig in further, take a look at the specification: `https://www.w3.org/TR/css-fonts-4/`.

However, before we bring this chapter to a close, there is one more font-related property I would like to bring your attention to that I have found very useful. That's `font-variant-numeric`.

Fancy numbers with font-variant-numeric

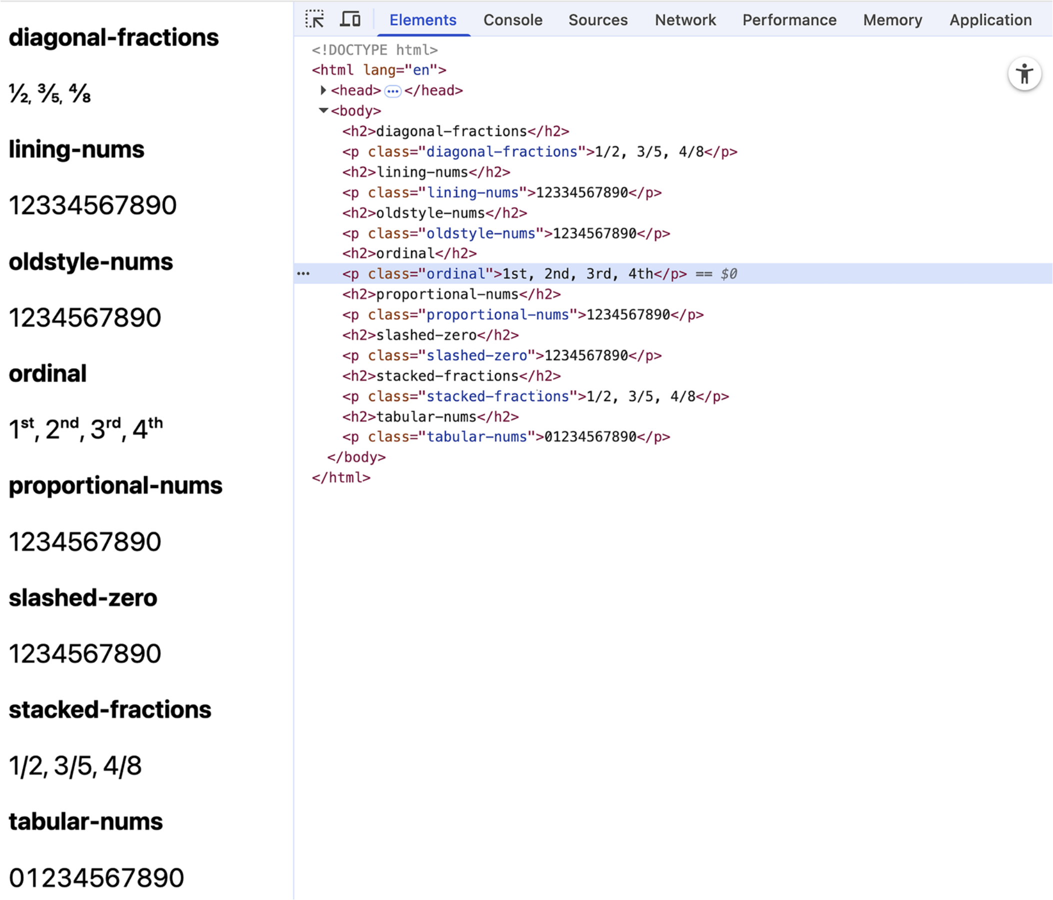
Aside from variable fonts, many modern fonts have variations on glyphs. Think about displaying a fraction, or whether ordinal markers such as 1st, 2nd, and 3rd are displayed with a superscript part, or whether a zero has a cross through it or not. You can make these choices by way of the `font-variant-numeric` property and associated value. You can reset the value with `normal`, such as `font-variant-numeric: normal`, or you can pass one or more of the following options:

- `diagonal-fractions`: Those little fractions, such as ¾
- `lining-nums`: Makes all numbers sit on the baseline
- `oldstyle-nums`: Makes sum numbers sit below the baseline

- `ordinal`: Provides special glyphs for 1st, 2nd, 3rd, and so on
- `proportional-nums`: Allows some numbers to be different sizes
- `slashed-zero`: Makes a zero have a bar through it
- `stacked-fractions`: Displays fractions above and below the line
- `tabular-nums`: Makes all numbers the same size

The one I have made most use of personally is `tabular-nums` as it means when building interfaces that have things such as timers, the times stay a consistent width as they change, which stops awkward shifts of surrounding elements.

Here is a grab from `example_07-05`, which has examples of each setting. Note that not all options work for each font. By default, the example uses a system font, but there is a Google font linked up (but commented out) that you can use instead to try different options:



diagonal-fractions

½, ⅗, 4⁄8

lining-nums

12334567890

oldstyle-nums

1234567890

ordinal

1st, 2nd, 3rd, 4th

proportional-nums

1234567890

slashed-zero

1234567890

stacked-fractions

1/2, 3/5, 4/8

tabular-nums

01234567890

```
<!DOCTYPE html>
<html lang="en">
 ▸<head> ··· </head>
 ▾<body>
    <h2>diagonal-fractions</h2>
    <p class="diagonal-fractions">1/2, 3/5, 4/8</p>
    <h2>lining-nums</h2>
    <p class="lining-nums">12334567890</p>
    <h2>oldstyle-nums</h2>
    <p class="oldstyle-nums">1234567890</p>
    <h2>ordinal</h2>
…   <p class="ordinal">1st, 2nd, 3rd, 4th</p> == $0
    <h2>proportional-nums</h2>
    <p class="proportional-nums">1234567890</p>
    <h2>slashed-zero</h2>
    <p class="slashed-zero">1234567890</p>
    <h2>stacked-fractions</h2>
    <p class="stacked-fractions">1/2, 3/5, 4/8</p>
    <h2>tabular-nums</h2>
    <p class="tabular-nums">01234567890</p>
  </body>
</html>
```

Figure 7.10: Different fonts support different numeric variations

Colored fonts

This is something we are not going to look at in depth here, as I think it is pretty niche, but worth relating nonetheless. There are a growing number of color fonts, plus some associated CSS properties such as `font-palette`. While fun, the file size of such fonts can be far greater than a normal font, so they aren't always the best choice if performance is key. For a good explainer, head over to `colorfonts.wtf`.

How much of that can you now understand? We have a system font stack, an `@font-face` rule loading a variable font (complete with ranges defined for weight and slant), and `sans-serif` as the default `font-family`, which we then immediately overwrite with `font-family` defined with a CSS custom property for browsers that understand it. Finally, we use a feature query for variable fonts and define settings for it, also adding in some additional font features we'd like to see.

Summary

In this chapter, we've looked in depth at web typography, considering the pros and cons of both system and web fonts, when each is most applicable, and the various properties and values we can apply to get the most from each.

We've also covered variable fonts and understood the potential and considerations for employing them in projects.

As you are now aware, there are fine details involved in wielding typography on the web to the best effect.

In the next chapter, we are going on another deep dive of a tenet of design on the web, and that's color. And just like web typography, I'll be surprised if there aren't more than a few things you didn't already know. That was certainly the case for me when I was writing it!

Exercise

Have a look at the CSS from `example_07-04` using the variable font:

```
:root {
    --MainFont: "Helvetica Neue", Helvetica, Arial, sans-serif;
}

@font-face {
    font-family: "Inter-V";
    src: url("fonts/inter.var.woff2") format("woff2-variations");
    font-weight: 100 900;
    font-display: fallback;
    font-style: oblique 0deg 10deg;
}

body {
    background-color: var(--background);
    color: var(--foreground);
    transition: all 0.35s;
    font-size: 1.2em;
    font-family: "sans-serif";
    font-family: var(--MainFont);
    font-weight: 400;
}

@supports (font-variation-settings: "wdth" 200) {
    body {
        font-family: "Inter-V";
        font-variation-settings: "wght" 300, "slnt" -4;
        font-feature-settings: "cv08", "cv05";
    }
}
```

8

CSS Colors

Color is one area of CSS that has enjoyed significant enhancements in the last few years. Previously, if you had an understanding of how to write a hex color, `#ff0000`, or the same color with the RGB color function, `rgb(255, 0, 0)`, or the HSL color function, `hsl(0 100% 50%)`, that was all the color knowledge required. Nowadays, you need to understand a whole slew of different formats and color functions to manipulate those formats, to consider yourself "up to speed."

The fine detail of color in CSS is not something you are likely to wrestle with in day-to-day responsive design work. Ordinarily, we take it for granted and paste in the colors from a design file with little thought. However, when you do need to manipulate or convert colors, or if you get asked to implement a color in P3, I don't want you looking embarrassed because I didn't explain these concepts to you. So, to repeat advice I have given elsewhere in the book, consider the possibilities presented here, but don't worry about remembering the syntax. That's what this chapter is for: to give you some guidance for that fateful day you need to implement something more involved with color.

Color theory is a dense and notoriously difficult subject. Not just conceptually but mathematically too. As such, we will concentrate almost entirely on the practical side of working with color in CSS. My aim here is to provide conceptual clarity, not necessarily pinpoint mathematical accuracy.

With that point made, let's look at what we will be covering in this chapter:

- RGB with hex notation
- RGB transparency with hex notation
- RGB functional notation
- HSL functional notation

- Space-separated functional color notation
- P3 color gamut
- The `@media (color-gamut: )` rule
- The `color()` function
- Lab and LCH
- How to write fallbacks for older browsers
- The `color-mix()` function
- The `color-contrast()` function
- Relative color syntax
- The `light-dark()` function

To begin our journey through the topic, we will start with a refresher on the basics of working with color in CSS before moving on to the newer color features and formats.

So, let's start with something you will likely already have a solid grasp of: RGB.

RGB color

RGB (**red**, **green**, and **blue**), like many of the formats we will look at, is a coloring system that's been around for decades. It works by defining different values for the red, green, and blue components of a color. For example, a red color might be defined in CSS as a hex (hexadecimal) value, `#fe0208`:

```
.redness {
    color: #fe0208;
}
```

The first two digits are the red value in hexadecimal, the next two are the green, and the last two are the blue.

For a great post describing how to understand hex values more intuitively, I can recommend this blog post at Smashing Magazine: `http://www.smashingmagazine.com/2012/10/04/the-code-side-of-color/`.

However, with CSS, that same color can equally be described with an RGB functional notation, where each value can be specified in a range from `0` to 255:

```
.redness {
    color: rgb(254, 2, 8);
}
```

Hex and RGB values are the kinds of values you will be used to seeing in graphics applications.

HSL color

Besides RGB, CSS also allows us to declare color values as **hue**, **saturation**, and **lightness** (**HSL**).

HSL isn't the same as HSB! Don't make the mistake of thinking that the **hue**, **saturation**, and **brightness** (**HSB**) value shown in the color picker of image editing applications such as Photoshop is the same as HSL—it isn't!

What makes HSL comparatively intuitive to use is that it's relatively simple to understand the color that will be represented, based on the values given. For example, unless you're some sort of color-picking ninja, I'd wager you couldn't instantly tell me what color `rgb(255, 51, 204)` is. Any takers? No, me neither.

However, show me the HSL value of `hsl(315, 100%, 60%)` and I could take a guess that it is somewhere between a magenta and a red color (it's actually a festive pink color). How do I know this? Simple: "Young Guys Can Be Messy Rascals!" This mnemonic will help you to remember the order of colors in an HSL color wheel, as you'll see shortly. HSL works on a 360° color wheel. It looks like this:

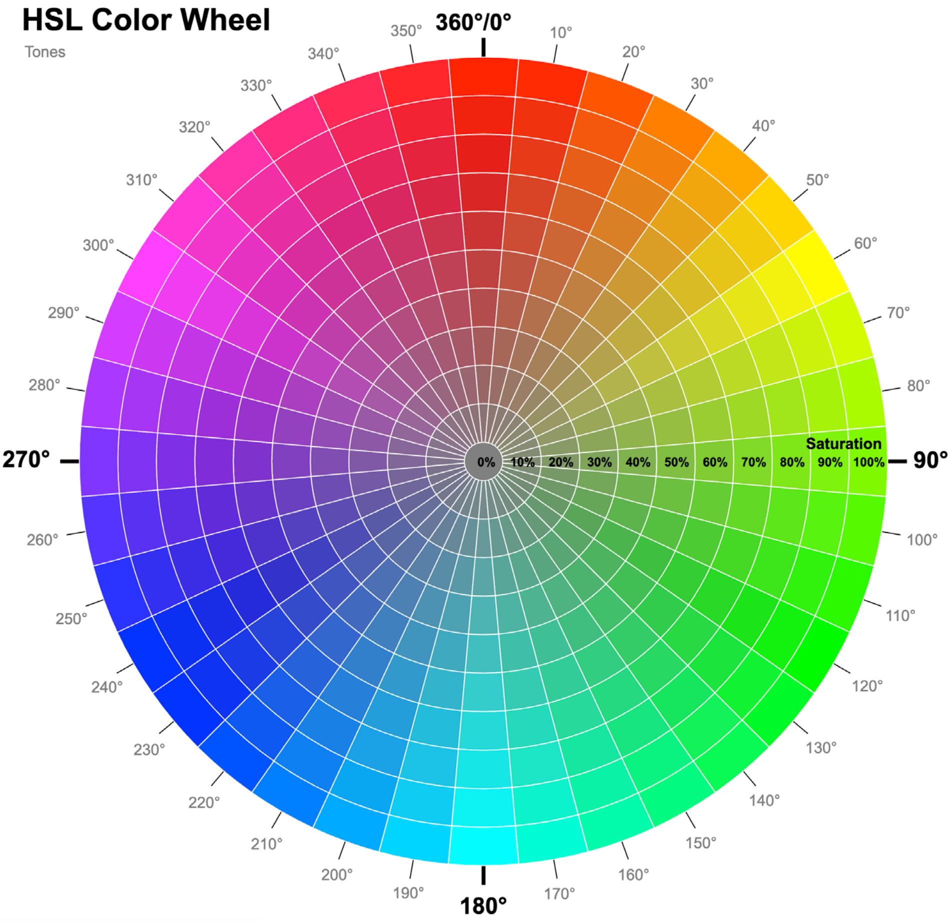

HSL Color Wheel
Tones
360°/0°
10°
20°
30°
40°
50°
60°
70°
80°
90°
100°
110°
120°
130°
140°
150°
160°
170°
180°
190°
200°
210°
220°
230°
240°
250°
260°
270°
280°
290°
300°
310°
320°
330°
340°
350°
Saturation
0%
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%

Figure 8.1: A HSL color wheel created with SVG (`https://codepen.io/websanity/pen/QWNMRq`)

The first figure in an HSL color definition represents **hue**. Looking at our wheel, we can see that yellow is at 60°, green is at 120°, cyan is at 180°, blue is at 240°, magenta is at 300°, and finally, red is at 360°. So, as the aforementioned HSL color had a hue of 315, it's easy to know that it will be between magenta (at 300°) and red (at 360°).

The next two values in an HSL definition are for **saturation** and **lightness**, specified as percentages. These merely alter the base hue. For a more saturated or "colorful" appearance, use a higher percentage in the second value. The final value, controlling the lightness, can vary between 0% for black and 100% for white. So, once you've defined a color as an HSL value, it's also easy to create variations of it, merely by altering the saturation and lightness percentages. For example, the same red color we defined in the *RGB color* section can also be defined in HSL values as follows:

```
.redness {
    color: hsl(359, 99%, 50%);
}
```

If we wanted to make a slightly darker color, we could use the same HSL value and merely alter the lightness (the final value) percentage value only:

```
.darker-red {
    color: hsl(359, 99%, 40%);
}
```

In conclusion, if you can remember the mnemonic "Young Guys Can Be Messy Rascals" (or any other mnemonic you care to memorize) for the HSL color wheel, you'll be able to approximately write HSL color values without resorting to a color picker, and also create variations upon it. Show that trick to the savant Rust, Node.js, and Zig developers at the office party and earn some quick kudos!

Alpha channels

It won't be long when working with designs on the web before you will be asked if the color of something can be partially transparent to allow some of what's beneath to show through. For that need, with hex, RGB, and HSL, it is possible to define an alpha channel.

Alpha with hex

With hex, you can define an alpha channel with a four-digit or eight-digit syntax. Here is black with a 50% alpha channel written first as a four-digit hex code:

```
background-color: #0007;
```

Then, here's the same color as an eight-digit hex code:

```
background-color: #00000077;
```

Two things to keep in mind with hex and transparency. Firstly, as the name tells you, it uses a hexadecimal numbering system, where 0-9 are handled with numbers, and 11 to 16 are handled with the "a" to "f" characters. So, imagine those characters in a line:

```
0123456789abcdef
```

You can see that halfway along is the character 7, which gives us 50% alpha.

The second thing to remember is that the four-digit hex doubles up each value. That's important if you want granularity because you can be more precise with two different digits for each color channel, which means in those scenarios, you will want to use the eight-digit variant.

It used to be necessary to define RGB or HSL colors with an alpha channel using the functional `rgba()` or `hsla()` formats (note the appended `a` for alpha). However, that's no longer needed unless you need to support a very old browser.

Alpha with functional notation

Using functional notation, you can define an alpha channel like this with HSL:

```
.redness-alpha {
    color: hsl(359, 99%, 50%, 0.5);
}
```

Or, by using a percentage if you like:

```
.redness-alpha {
    color: hsl(359, 99%, 50%, 50%);
}
```

The RGB syntax follows the same convention as the HSL equivalent:

```
.redness-alpha-rgba {
    color: rgb(254, 2, 8, 0.8);
}
```

The alpha is always represented with either a percentage or a numeric value between 0 and 1.

You may be wondering—why not just use opacity, as CSS allows elements to have opacity set with the `opacity` declaration? When we apply opacity to an element, a value is set between zero and one in decimal increments (for example, opacity set to `0.1` is 10%). However, opacity differs from an alpha channel. Setting an opacity value on an element affects the *entire* element, whereas setting a value by using an alpha channel allows particular *parts* of an element to have an alpha layer. For example, an element could have an alpha value for the background color but a solid color for the text within it.

OK, that brings us to the beginning of the newer CSS color capabilities.

In the last code snippet, we defined an RGB color with an alpha channel using this kind of syntax:

```
.redness-alpha-rgb {
    color:  rgb(254, 2, 8, 0.8);
}
```

However, we can now write things space-separated instead.

Space-separated functional color notation

Unless you need to support old browsers, you can, and probably should, write colors using the newer **space-separated** syntax. That doesn't get you anything extra for the sRGB-based notations we have already looked at, but as the newer color formats, which we will look at momentarily, also use the space-separated syntax, you may as well just use the same style for all.

So, space-separate the color values, and then if you need an alpha channel, add a forward slash (/) before adding the alpha as either a decimal value or a percentage.

Here's an example:

```
.redness-alpha-rgba {
    color: rgb(254 2 8 / 0.8);
}
```

Or, this if you prefer percentages:

```
.redness-alpha-rgba {
    color: rgb(254 2 8 / 80%);
}
```

So, now you know the syntax we will be using for all the new color formats, let's cover some groundwork before taking an in-depth look.

Color concepts and terminology

With the formats we have looked at so far, it has not been necessary to burden you with too much color background to make practical use of the techniques and syntax we have been using. As we look at the newer formats, syntaxes, and capabilities, we will likely need to extend our "palette" (pun intended) of information in order to make better sense of it all.

Therefore, before we get into things proper, it would serve us well to define a couple of terms we will be using throughout the remainder of the chapter.

Color space

Different color spaces exist to describe varying amounts of color. In the print world, most are familiar with **CMYK** color space, where cyan, magenta, yellow, and black are mixed to create the colors you wish to see printed. With correctly calibrated equipment, the color you see onscreen is the color you will see printed. For us workers in the web world, **sRGB**, the standard color space on the web, is the color space that we are most familiar with (created by HP and Microsoft in 1996 for computer monitors). It is a system that lets us specify colors as varying amounts of red, green, and blue.

From a technical perspective, a color space is a mathematical representation of colors. It is a way of plotting, as numbers, using varying parameters, all the possible colors that a given color space can represent.

As we will see momentarily, CSS now lets us define colors in different color spaces too.

Gamut

For our purposes, when we talk about **gamut**, we are typically talking about the range of possible colors that a device can display. A modern display can display more colors than the displays of yesteryear. The following figure shows the comparative gamuts of a few color spaces. Notice that the two we are primarily interested in for this chapter, sRGB and P3, are far smaller than the others.

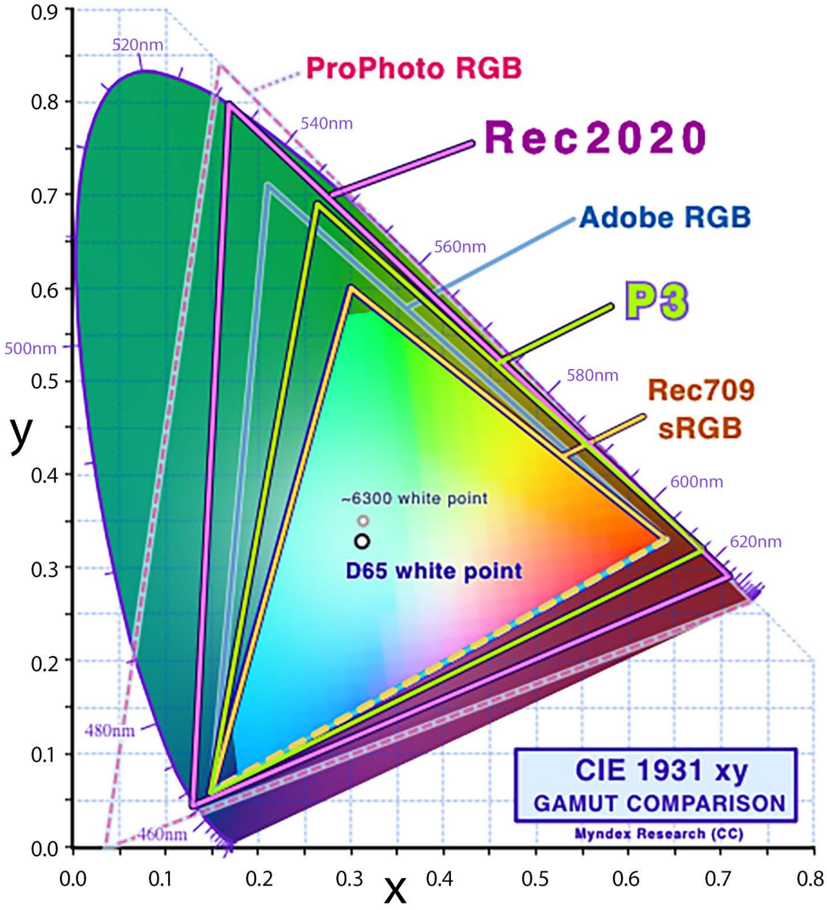


Figure 8.2: This image from Wikipedia shows how little sRGB represents when compared with P3 and other color spaces

All you need to appreciate for the understanding of this chapter is that there are many devices these days that are capable of displaying colors that our trusty RGB notations cannot express.

Larger gamuts are becoming more and more of a selling point with hardware. To exemplify, here is the paragraph from Dell's technical specifications describing the colors of the monitor I am looking at (`https://www.dell.com/en-uk/shop/dell-ultrasharp-40-curved-wuhd-monitor-u4021qw/apd/210-ayjf/`):

> *See true-to-life color with an incredible color depth of 1.07 billion colors and wide color coverage across industry standards, including 100% sRGB, 100% Rec. 709 and 98% DCI-P3. DCI-P3 (which offers nearly 25% more coverage than sRGB), enables a wider range of colors and is quickly becoming the color standard for creating and viewing content.*

"Wide color coverage" and "wide gamut" are the kinds of phrases that would indicate you are looking at a screen that can deal with colors (somewhere between 25% and 50% more) you may never have yet seen it display. Probably not with CSS, anyway. Well, we will address that shortly!

> If you want to go deeper into this color theory, I recommend the following post on the WebKit blog from 2016 that I keep coming back to time and again: `https://webkit.org/blog/6682/improving-color-on-the-web/`. Don't let the comparatively old date put you off; much of the color work we do these days was figured out in the 1940s and 1950s!

Now we have considered the notion of color space as a mathematical expression of color, and gamut as the range of possible colors a device can display, I think we are finally ready to look at what's new with CSS color.

The current state of CSS color

As I write this book in 2025, here is the briefest summary this author can manage of the state of CSS color:

- All monitors and browsers can display sRGB colors, and there are various syntaxes to describe them in CSS – hex (for example, `#000`), `rgb(0,0,0)`, and `hsl(0 0% 0%)`.
- There are also colors that many modern monitors can display, but only newer browser versions support. This category of colors is typically dealt with in CSS with the `color()` function to pass a custom color space (typically `display-P3`) to CSS, or using LCH/Lab.
- Finally, there is a category of colors that humans can see, yet few (if any?) monitors can display. CSS can now also describe such colors with LCH/Lab.

Introductory note

As we work through all these formats, be aware that there is no real "best" color format. They are all useful for certain applications. For example, I tend to use tried and tested hex values the majority of the time; they are the values that designers almost always supply me with, and there is no value in converting them to another format. I tend to make use of one of the newer formats when I need to realize a certain style of gradient, for example, or use a color that simply can't be expressed with a hex or `rgb()`. Hopefully, the notion that there is no "best" format, only a "best format for this job," will become obvious as we work through the chapter.

> As we are concentrating on the practicalities of color in CSS, you should be aware that there are other color **formats** or syntaxes that may be useful for a specific use case but that we don't cover here. HWB, for example, can also describe sRGB colors and uses a **Hue, Whiteness, Blackness** syntax. I have never found a need to use **HWB**, but that doesn't mean you won't! More on HWB here: `https://www.w3.org/TR/css-color-4/#the-hwb-notation`.

Right, let's get on and show you those colors you have never seen on your screen before.

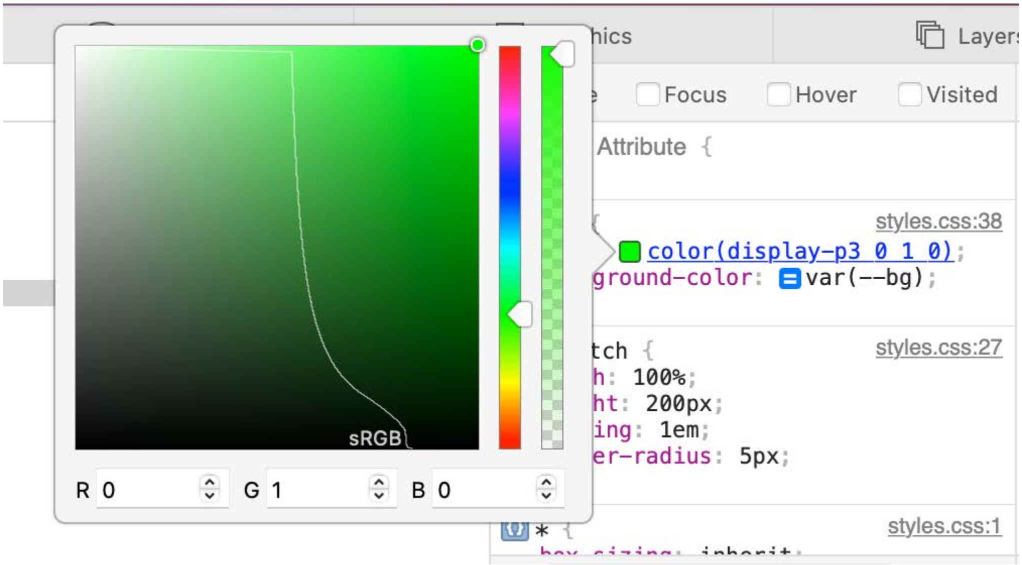
Display-P3

P3 is a superset of sRGB and makes up to 50% more colors possible. It's the vivid end of the red, green, and blue colors that benefit the most, particularly the green.

Before I get you too excited, a caveat. You'll need to be using a modern enough screen to see these colors, or this is going to be a letdown on a par with the end of the *Game of Thrones* TV series.

> It is also almost redundant to point out that the images in this book will also be unable to convey the colors accurately. You will either have to trust me or get that browser open and see for yourself.

OK, open `example_08-01` and inspect the page with your development tools (in Safari, I'm doing **Develop | Show Web Inspector**). Select the first color "swatch" and open the color picker up for `background-color`. What do you notice?



```
Attribute {
{
🟩 color(display-p3 0 1 0);
ground-color: var(--bg);
tch {
h: 100%;
ht: 200px;
ing: 1em;
er-radius: 5px;
* {
```

Figure 8.3: The Safari color picker shows where sRGB ends and P3 takes over

In Safari, in the color picker tool, I see a representation of the standard sRGB range of colors, but the color we have defined is beyond the sRGB confines we are used to seeing and into the area that only the `display-p3` color space can display. I hope your retinas are just a little shocked at the vividness on show!

Because it is a color space we are using, as opposed to a different color model, we use the `color()` function to pass this predefined color space in, and then set the red, green, and blue values. So, if we want the greenest green that P3 has to offer, we can set it in CSS like this:

```
background-color: color(display-p3 0 1 0);
```

It differs only in that, rather than using 0-255 to set each color channel, it uses a decimal between 0 and 1, where `0` is none of that channel and `1` is all of it.

> If you don't have access to a browser or hardware that can display this color space, then the following post, also on the WebKit blog, offers some good examples, with comparison pictures of sRGB and P3: `https://webkit.org/blog/10042/wide-gamut-color-in-css-with-display-p3/`.

Our prior example looks like this in code:

```
color(display-p3 0.135 1 0);
```

Browsers that support P3 should have a color picker capable of playing with it, so you can see as you play with the colors how the values express the different amounts of red, green, and blue. Try another color that plays to the strengths of P3:

```
color(display-p3 0 1 1);
```

That will get you a very vivid cyan color. P3 is great for these kinds of bright colors, and simple to work with, with just a basic understanding of red, green, and blue.

> Browsers that do support wider gamuts such as `display-p3` can also display images that have a relevant color profile attached – an oft-overlooked fact.

The @media (color-gamut) media query

Although we dealt with media queries in great detail in *Chapter 3*, there is one media query better dealt with here. That is the `color-gamut` media query, which lets us test for the support of certain colors on the host device.

Here is the syntax. In this instance, we are checking whether a device is capable of supporting the `rec2020` gamut, a gamut that includes P3 and more. There is no relevance here other than for illustration.

```
@media (color-gamut: rec2020) {
    .thing {
        color: color(display-p3 0 1 1);
    }
}
```

I don't imagine this is a feature most developers will find themselves needing regularly, if at all, but it is mentioned here just in case.

LCH with the lch() function

Even the larger gamut of P3 is nowhere near the limits of human visual perception. However, we now have a format in CSS capable of doing that. Lab/LCH are designed to be able to express all of human vision, even if, currently, our monitors are not capable of it.

LCH is an abbreviation of **Lightness**, **Chroma**, **Hue**. It is the same color space as Lab but uses polar coordinates.

> If you aren't familiar with the term **polar coordinates**, it's a way to describe how far something is from a reference point. Imagine looking down on a color wheel that shows all the possible colors of a color space. The center of that wheel is the pole, our reference point. You can describe where one color is in relation to the reference point by giving a direction/angle and a distance along the axis of that angle. That is the essence of how polar coordinates work in CSS color and mathematics.

The syntax is lightness as a percentage, then two numerical values: one for the chroma, and the third value for the hue angle. So, here is an example of a very bright red:

```css
background-color: lch(50% 132 43);
```

Like the other functional color notations, you can add a forward slash (/) and then add an alpha channel as a value between 0 and 1 or a percentage.

Arguably, one of the more important things to take from LCH/Lab is that the lightness is designed to look the same across different colors, something that the lightness of HSL does not do. So, you should be able to define two different colors with the same lightness in `lch()`, and they should appear to have the same perceptual lightness.

If you compare swatch 3, "`hsl()` same lightness", and swatch 4, "`lch()` same lightness", in the example file, hopefully you can appreciate that the lightness of the two colors in the LCH example is more similar than the lightness of the two HSL colors.

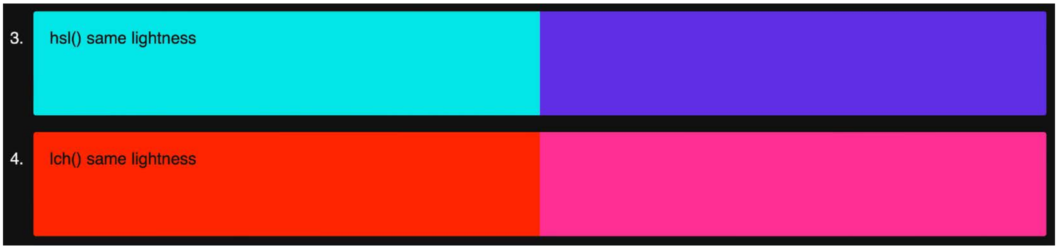


Figure 8.4: Comparison of two colors in HSL (top) and LCH (bottom) that should have the same lightness

Practically, when working with LCH, lower the lightness value for a darker variant of a color, and increase it for a lighter one.

There are some points to be aware of when using `lch()`. Firstly, you may, as I did, incorrectly assume that, as it uses a wheel for the hue, then, like HSL, the wheel would be the same one as for HSL. It isn't!

Consider our swatch with the gradient defined with one side as `lch()` and then those values mapped to what you might think would be the equivalent positions in the `hsl()` section (this is swatch 8 in `example_07-01` for the chapter):

```
.lch-hsl {
    background: linear-gradient(
        to right,
        lch(50% 150 45) 50%,
        hsl(45 100% 50%) 0
    );
}
```

Here is what that looks like:

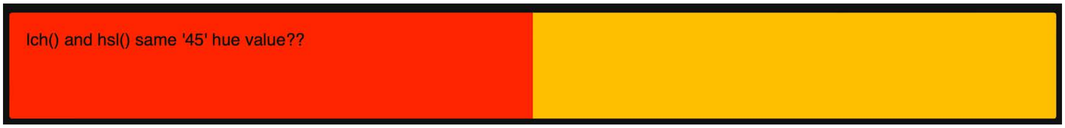


Figure 8.5: Setting lch and hsl to what you think should be the same color does not produce the same color!

Hmmmm, nothing is ever easy when it comes to color conversion, I am afraid! Those two look nothing like one another. We would need to remember our earlier HSL mnemonic and move our HSL hue to `0` to make it similar to the position of red in the wheel used by LCH.

> If you are wondering why the gradient defined gives us a hard stop, we cover that in the next chapter, *Stunning Aesthetics with CSS*.

I found the Chroma part of the `lch()` syntax hard to conceptualize initially, as the maximum value of color intensity not only differs between colors and screen gamut, but it is also theoretically limitless. Luckily, an out-of-range value gets normalized in the browser, so if you are after an intense color, just use a high value (say, `150`) and the browser will set it accordingly. Whenever I set a value for the chroma like that, it reminds me of Spinal Tap's "These go to 11!" (kids, ask your parents, or I'll let you search on YouTube for it if you promise to come right back).

For more mathematical clarity on Lab and LCH, head over to `https://www.w3.org/TR/css-color-4/#lab-colors`.

OKLab and OKLch

Smart color people recognized some shortcomings with Lab and LCH. In late 2020, Bjorn Ottosson introduced us to OKLab, humorlessly titled as such because it is an "OK Lab color space"! You can read all about the background and rationale in this blog post: `https://bottosson.github.io/posts/oklab/`.

Despite its relative newness, it has already made its way into the CSS Color Module Level 4 specification and gets its own polar form `oklch()` and `oklab` functional notations: `https://www.w3.org/TR/css-color-4/#ok-lab`.

In CSS usage, it seems that most people will find it useful for "better" and smoother gradients.

The specification notes that:

> *It was produced by numerical optimisation of a large dataset of visually similar colors, and has improved hue linearity, hue uniformity, and chroma uniformity compared to CIE LCH.*

From my brief experimentation with LCH versus OKLab gradients, I couldn't see an awful lot of difference. Swatches 6 and 7 in the example file show two gradients, the first done with `lch()` and the next with `oklab()`, both using comparable colors from the specification examples. To my eyes, on my screen, I couldn't really see much difference. Your mileage may vary!

Fallbacks for older browsers

There is a nice, simple way of providing a fallback for older browsers – simply write the standard format first, and then the newer syntax straight after.

So, let's say we wanted that "super green" P3 green, but know that we have a nice non-P3 version for browsers and hardware that don't support P3 (note I'm also using the older syntax with commas for the sake of even older browsers):

```
background-color: rgb(0, 255, 0);
background-color: color(display-p3 0 1 0);
```

It doesn't get much easier than that. Of course, we could also use `@supports`, which we looked at in detail in *Chapter 6*, *CSS Nesting, Layers, Selectors, and More*.

```
@supports (color(dispay-p3 0 1 0)) {
    background-color: color(display-p3 0 1 0);
}
```

At first glance, the `@supports` version may seem more verbose. It is, until you start passing color values around as custom properties instead.

We cover custom properties fully in *Chapter 13*, *Custom Properties and CSS Functions*, but for now, all you need to understand is that, in this instance, we are using a custom property as a means of storing a value that we can reuse in multiple places.

We would set up a color to reuse like this:

```
:root {
    --col-accent: rgb(0 255 0);
}
@supports (color: color(display-p3 0 1 0)) {
    :root {
        --col-accent: color(display-p3 0 1 0);
    }
}
```

And then, whenever we wanted to use that color, we would only have to write the following:

```
.thing {
    color: var(--col-accent);
}
```

The user would then get either the `rgb()` version or the `display-p3` version, depending on their system. With it set as a custom property, we can reuse it as many times as we like in that manner and know that each user will get the right syntax for their environment.

The color-mix() function

This is another new function for color.

As ever, check support for features at caniuse.com (`https://caniuse.com/?search=color-mix()`).

Consider swatch 10 in our example. You should see a color that is like a mix between gold and orange, because that is exactly what it is!

Here, we are using the `color-mix()` function to mix `gold` and `orange`:

```
.mix {
    background-color: color-mix(in lch, gold 50%, orange);
}
```

The `color-mix()` function lets you specify a color space to mix in, and then two colors to mix in that color space. It's like kindergarten for web developers!

Here are the essentials to understand. You pass a percentage after the first color (and you can use any of the color notations we have looked at in this chapter), and that is the amount of that color passed into the "mixer." The percentage left is given to the second color.

It's early days for `color-mix`, but I have yet to find any compelling use cases for it. However, it's an interesting addition to our color capabilities, so hopefully you'll find a use case for it.

You can read the specification for `color-mix()` here: `https://www.w3.org/TR/css-color-5/#color-mix`. It also gives in-depth detail on how the gamuts are mapped when values are passed in that the mixing color space cannot deal with, and lots more.

The contrast-color() function

I felt the `contrast-color()` function was going to be very useful when it was the `color-contrast()` function, but not anymore. Confused? I'm not surprised. Let me explain.

The original color-contrast() function

In the prior version of this book, I detailed a new function that helped to choose contrasting colors, as it was specified at that time. The function was `color-contrast()` and it let you specify a color, then at least another two colors, to effectively compete for the title of "most contrasting color," like this:

```
.header-text {
    color: color-contrast(#333 vs #666, #545454, #777, #f90);
}
```

You can see two examples of `color-contrast()` being used in the code of swatches 11 and 12 of our example. However, you may not actually be able to see them, as they are no longer working in the latest browser versions. Why?

The original color-contrast() function was flawed

That `color-contrast()` function has now been removed from the specification and replaced with the (honestly, not confusingly named at all) `contrast-color()` function, which, sadly for this author, is about as much practical use right now as a chocolate teapot. Right now, all `contrast-color()` does is return black or white given an input. That's it. Womp womp. How did we arrive at this travesty?

How did this happen?

The short version is the WCAG 2.0 contrast algorithm used in the prior function, `color-contrast()`, was simply not very good. It would often produce "contrasting" colors that simply did not contrast well given the input. It would sometimes choose colors that were mathematically more contrasting but not perceptually more contrasting. That is obviously a bit of a problem, so `color-contrast()` was asked to leave the building.

Instead, we have the functionally inferior, but more often "correct" `contrast-color()`. The promise is that, at some point, some of the useful abilities of the prior function may make their way into this newer function. But for now, we can enjoy, well, very little as it happens.

The newer, but more limited contrast-color() function

Let me tell you how this works, and hopefully you can share in my frustration. You pass it any color, and it returns as either black or white. It looks like this:

```css
.contrast-color {
    --bg: #f90;
    background-color: var(--bg);
    color: contrast-color(var(--bg));
}
```

So here, we set a custom property to an orange color, and then the background color of the swatch to that custom property. Then, we pass that same orange custom property to the `contrast-color()` function, and it returns black in this example, because black contrasts better than white against that orange color.

Because you can only get black or white out of the `contrast-color()` function currently, I don't think I will find a use case for it. It's incredibly limited, and as such, it isn't really something I can recommend. Hopefully, future iterations in browsers will offer improvements.

> There is a more in-depth write-up of the newer `contrast-color()` function and why the web has ended up with it on the WebKit blog: `https://webkit.org/blog/16929/contrast-color/`.

Relative colors with the relative color syntax

One color function I do use regularly, and I am excited to tell you about because I use it all the time, is the relative color syntax. This lets you amend a color by kind of deconstructing it. Consider swatch 13 in our example file, which is set with this CSS:

```
.relative {
    --bg: #f90;
    background-color: hsl(from var(--bg) h s calc(l + 10));
}
```

Admittedly, that can look quite daunting. Let's break it down. So, here we have a custom property set to a hex value (`#f90`). Then, with our `hsl()` color function notation, we use the `from` keyword to "pick" where we are generating our relative color from. In this case, it is the hex defined as a custom property in the preceding line, but it could be any color set in any of the notations that CSS allows. Then, we can use the letters of the color type – in this case, `h`, `s`, and `l` – to get the component parts of the original color into our current format and then tweak it. In this instance, I have used the `calc()` function to lighten the color by 10.

Let's look at one more. This is swatch 14:

```
.relative-rgb {
    --bg: hsl(0 100% 50%);
    background-color: rgb(from var(--bg) r 100 b / 80%);
}
```

Here, we take a color defined in our custom property in HSL, then tweak the green value to be `100` and add an 80% alpha layer.

Values always have to be numbers

Do you remember that in HSL, lightness is always a percentage? Initial implementations of the relative color function allowed you to use percentages in the calc expressions – for example, `calc(l + 10%)`, but that does not work in the current implementation supported across browsers. The value being added or subtracted always needs to be a number.

This also holds true when specifying the alpha channel, or transparency of a color, which can be expressed ordinarily with a percentage or decimal, such as 80% or .8.

If manipulating the alpha channel, use the keyword alpha and manipulate it with decimal values. For example, consider swatch 15 from the example file:

```
.relative-rgb-alpha {
    --bg: hsl(0 100% 50% / 45%);
    background-color: rgb(from var(--bg) r 100 b / calc(alpha + 0.5));
}
```

Here, we are using an hsl color, moving it into RGB for the manipulation, using alpha in the `calc()` function, and adding `.5` (or 50%) to alpha (set in the custom property to 45%).

The relative color syntax is a wonderfully useful addition to CSS. It means, as a developer, we can easily tweak colors programmatically. When we are asked to "just make that color a little more saturated on hover" or similar, we can easily do that, faster, and with more confidence than ever before.

The light-dark() function

If you find yourself needing to support different colors depending upon whether the user has their device set to light or dark mode, the `light-dark()` function is here to make your life easier. It allows you to provide for both eventualities for a property with a single function.

First, I need to say that, in order to use `light-dark()`, you need to opt in by adding this to the root of your page:

```
:root {
  color-scheme: light dark;
}
```

The `color-scheme` property can be set to `light`, `dark`, both of those values, as we have above, or just one with the `only` keyword, such as `only light`.

The `color-scheme` property can be applied to individual elements, but I always find it easier to simply set it on the root, which means that everything is opted in.

With that set, with any element, you can set the light and dark colors using the `light-dark()` function like this.

```
.thing {
    background-color: light-dark(#eee, #222);
    color: light-dark(#222, #eee);
}
```

Here, the first value is the color applied if the color scheme is set to "light" mode, and the second is the value that gets set if set to "dark" mode. It's as simple as that. No more need for a `prefers-color-scheme` media query, just to set a different color value depending on whether the user preference is light or dark mode.

Summary

In CSS terms, color capability is advancing rapidly. We looked at the basics of setting colors and alpha layers in CSS using hex and functional notation. We also considered how to make use of wider gamuts such as `P3` and how to provide fallbacks in a couple of ways for older devices.

Later, we looked at the cutting-edge features of color. We looked at mixing colors with `color-mix()`, creating a color relative to another with the relative color syntax, and we considered the shortcomings of the `contrast-color()` function.

As this is such a dense topic, the following is a list of resources to read further on the topic. But that doesn't mean we won't look at color more. On the contrary, we will need it in the next chapter, *Chapter 9*, *Creating Stunning Effects with CSS*, where we will be looking at gradients, shapes, and so much more.

Resources

- An introduction to the OKLab color space: `https://bottosson.github.io/posts/oklab/`
- Great arguments for LCH: `https://lea.verou.me/2020/04/LCH-colors-in-css-what-why-and-how/`
- The latest CSS color specification: `https://www.w3.org/TR/css-color-5/`
- A video from Chris Lilley from the W3C explaining gamut reduction with OKLab: `https://www.youtube.com/watch?v=dOsp6u4bIwI`
- A solid explainer on wide gamut color: `https://webkit.org/blog/10042/wide-gamut-color-in-css-with-display-p3/`
- An excellent and accessible overview of color in CSS: `https://css-tricks.com/new-css-color-features-preview/`

9

Stunning Aesthetics with CSS

In this chapter, we are going to look at a selection of CSS capabilities for enriching your designs. Every year that passes, CSS introduces more features that enable us to do in code what was previously only possible with an image editor.

Replacing images with effects in code is usually a good thing. Not only does it tend to make things more maintainable and flexible, but it also results in less page "weight" for the end user, with code typically producing a far smaller file size than the comparable image.

In this chapter, we will cover the following:

- How to create text shadows
- How to create box shadows
- How to make linear and radial gradients
- How to make conic gradients
- How to use multiple backgrounds
- Using CSS background gradients to make patterns
- How to implement high-resolution background images with media queries
- How to use CSS filters (and their performance implications)
- Clipping with clipping paths
- Making use of the `shape()` function
- Masking elements with image masks
- Mixing the colors of elements with `mix-blend-mode`

These are by no means the only things possible in CSS, but they are a selection of things I have found most useful when building responsive websites and applications for the past decade. Let's dig in.

Text shadows

Let's make a start by looking at text shadows. Text shadows are a fairly simple way to change the aesthetics of text, and therefore provide a good starting point. Support for `text-shadow` is also ubiquitous. Let's first consider the basic syntax:

```css
.element {
  text-shadow: 1px 1px 1px #ccc;
}
```

The first value is the amount of shadow to the right, the second is the amount down, the third value is the amount of blur (the distance the shadow travels before fading to nothing), and the final value is the color. Shadows to the left and above can be achieved using negative values as the first two values. Here is an example:

```css
.text {
  text-shadow: -4px -4px 0px #dad7d7;
}
```

The color value doesn't need to be defined as a hex value. It can just as easily be HSL(A) or RGB(A):

```css
text-shadow: 4px 4px 0px hsl(140 3% 26% / 0.4);
```

You can also set the shadow values in any other valid CSS length units, such as `em`, `rem`, `ch`, and so on. Personally, I rarely use `em` or `rem` units for `text-shadow` values. As the length values tend to be low, using 1px or 2px generally looks good across all viewports.

Thanks to media queries, we can easily remove text shadows at different viewport sizes, too. The key here is the `none` value:

```css
.text {
  text-shadow: 2px 2px 0 #bfbfbf;
}
@media (min-width: 30rem) {
  .text {
    text-shadow: none;
  }
}
```

It's worth knowing that, in CSS, where a value starts with a zero, such as `0.14s`, there is no need to write the leading zero: `.14s` is exactly the same.

If there is no blur to be added to a text shadow, the value can be omitted from the declaration. Here is an example:

```
.text {
  text-shadow: -4px -4px #dad7d7;
}
```

That is perfectly valid. The browser assumes that the first two values are for the offsets if no third value is declared.

Multiple text shadows

It's possible to add multiple text shadows by comma-separating two or more shadows, as in this example:

```
.multiple {
  text-shadow: 0px 1px #fff, 4px 4px 0px #dad7d7;
}
```

Also, as CSS is forgiving of whitespace, you can lay out the values like this if it helps with readability:

```
.text {
  text-shadow:
    3px 3px #bbb, /* right and bottom */
    -3px -3px #999; /* left and top */
}
```

You can read the W3C specification for the `text-shadow` property here: `https://www.w3.org/TR/css-text-decor-3/#text-shadow-property`.

So, that's how you create shadows around text. What about when you want a shadow on a containing element?

Box shadows

Box shadows allow you to create a box-shaped shadow around the outside or inside of an element. Once you understand text shadows, box shadows are a piece of cake. Principally, they follow the same syntax: horizontal offset, vertical offset, blur, spread (we will get to spread in a moment), and color. Only two of the four length values are required. In the absence of the last two length values, a value of zero is assumed. Let's look at a simple example:

```
.shadow {
  box-shadow: 0 3px 5px #444;
}
```

The default `box-shadow` is set on the outside of the element. Another optional keyword, `inset`, allows the box shadow to be applied inside the element.

Inset shadow

The `box-shadow` property can also be used to create an inset shadow. The syntax is identical to a normal box shadow, except we add the keyword `inset`. I always add it at the beginning, but it doesn't have to be there:

```
.inset {
  box-shadow: inset 0 0 40px #000;
}
```

Everything functions as before, but the `inset` part of the declaration instructs the browser to set the effect on the inside. If you look at `example_09-01`, you'll see an example of each type:

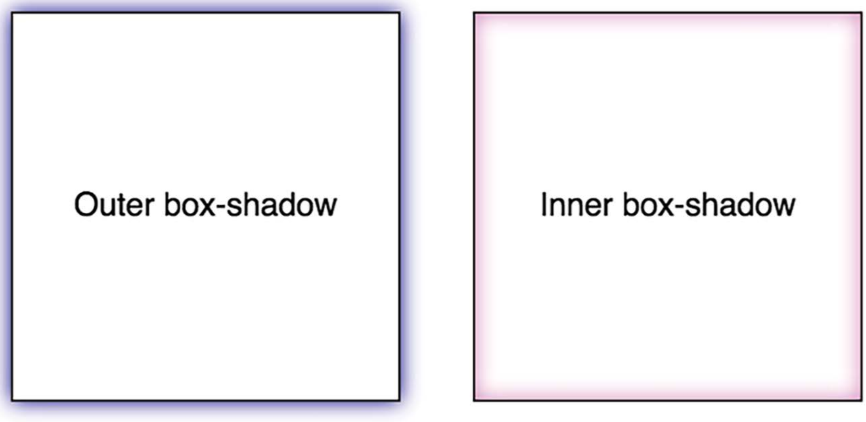


Figure 9.1: Outer shadows and inner shadows are both easily achievable

Multiple box shadows

Like text shadows, you can apply multiple box shadows. Separate the `box-shadow` declarations with a comma. They are applied from bottom to top (last to first) as they are listed. Remind yourself of the order by thinking that the declaration nearest to the top in the rule (in the code) appears nearest to the "top" of the order when displayed in the browser. As with `text-shadow` declarations, you may find it useful to use whitespace to visually stack the different `box-shadow` declarations:

```
box-shadow:
  inset 0 0 30px hsl(0deg, 0%, 0%),
  inset 0 0 70px hsl(0deg, 97%, 53%);
```

Stacking longer, multiple values one under the other in the code has an added benefit when using version control systems; it makes it easy to spot differences when you compare, or **diff**, two versions of the same file. It can be useful to stack selectors one under the other for that same reason.

Understanding spread

I'll be honest, for years, I didn't truly understand what the `spread` value of `box-shadow` actually did. I don't think the name "spread" is useful. Think of it more as an offset. Let me explain.

Look at the box on the left in `example_09-02`. This has a standard box shadow applied, with no spread. The one on the right has a negative `spread` value applied. It's set with the fourth value. Here is the relevant code that deals with each shadow:

```
.no-spread {
  box-shadow: 0 10px 10px;
}
.spread {
  box-shadow: 0 10px 10px -10px;
}
```

Here is the effect of each (element with a `spread` value on the right):

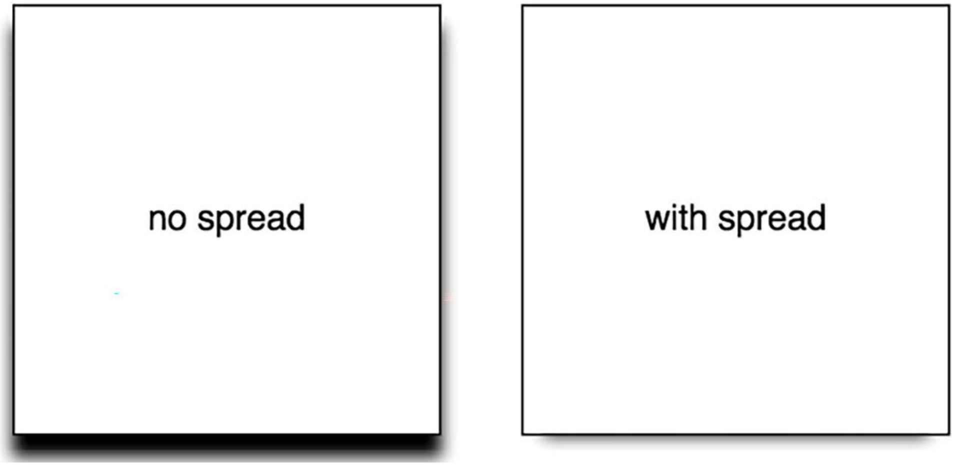


Figure 9.2: With spread, you can control how much shadow leaks out

The `spread` value lets you extend or contract the shadow in all directions by the amount specified. In this example, a negative value is pulling the shadow back in all directions. The result is that we can just see the shadow at the bottom of the right example, instead of seeing the blur "leak" out on all sides.

> You can read the W3C specification for the `box-shadow` property here: `https://www.w3.org/TR/css-backgrounds-3/#the-box-shadow`.

Right, so that's shadows. We will go on to deal with more shadows when we look at `drop-shadow` as part of CSS filters. For now, let's move on to gradients.

Background gradients

In days gone by, to achieve a background gradient on an element, it was necessary to tile a thin, graphical slice of the gradient. It was a pain to tweak as it meant round trips into a graphics application, and then, when a site was live, you would often experience a flash of unloaded gradient while the background image was fetched.

Thankfully, such a hassle is now nothing more than a distant memory for me; with a CSS `background-image` gradient, things are far more flexible. CSS enables us to create linear, radial, and conic background gradients, and repeating versions of each. Let's look at how we can define them.

The specification for CSS Images Module Level 3 can be found at `https://www.w3.org/TR/css-images-4/`.

The linear-gradient notation

The `linear-gradient` notation, in its simplest form, looks like this:

```
.linear-gradient {
  background: linear-gradient(red, blue);
}
```

This will create a linear gradient that starts at red and fades to blue. It will go from the top to the bottom unless you specify otherwise.

Specifying gradient direction

If you want to specify a direction for the gradient, there are a couple of ways to do this. The gradient will always begin in the opposite direction to where you are sending it.

Here is an example:

```
.linear-gradient {
  background: linear-gradient(to top right, red, blue);
}
```

In this instance, the gradient heads to the top right, so it starts red in the bottom-left corner, fading to blue at the top right.

If you're more mathematically minded, you may believe it would be comparable to write the gradient like this:

```
.linear-gradient {
  background: linear-gradient(45deg, red, blue);
}
```

However, keep in mind that on a rectangular box, a gradient that heads "to top right" (always the top right of the element it's applied to) will end in a slightly different position from `45deg` (always 45 degrees from its starting point).

It's worth knowing that you can also start gradients before they are visible within a box, as in this example:

```
.linear-gradient {
  background: linear-gradient(red -50%, blue);
}
```

This would render a gradient as if it had started before it was even visible inside the box, which is useful if you want your gradient to begin with "the end" of the fade from one color.

We actually used a color stop in the previous example to define a place where a color should begin and end, so let's look at those more fully.

Color stops

Perhaps the handiest thing about background gradients is **color stops**. They provide the means to set which color is used at which point in a gradient. With color stops, you can specify something as complex as you are likely to need. Consider this example:

```
.linear-gradient {
  margin: 1rem;
  width: 400px;
  height: 200px;
  background: linear-gradient(
    #f90 0,
    #f90 2%,
    #555 2%,
    #eee 50%,
    #555 98%,
    #f90 98%,
    #f90 100%
  );
}
```

Here's how that `linear-gradient` renders:

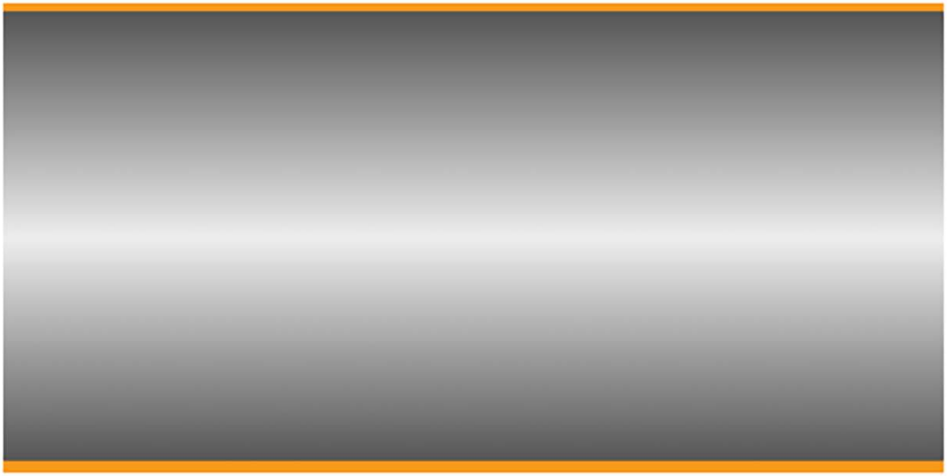


Figure 9.3: You can add as many stops as you like to a linear gradient

In this example (`example_09-03`), a direction has not been specified, so the default top-to-bottom direction applies.

Color stops inside a gradient are written comma-separated and defined by giving first the color, and then the position of the stop. It's generally advisable not to mix length units in one notation, but you can. You can have as many color stops as you like, and colors can be written as a keyword, hex, RGB, HSL, or any other color format we looked at in the previous chapter.

You can read the W3C specification for linear background gradients at `https://www.w3.org/TR/css-images-3/#linear-gradients`.

Having covered linear background gradients, let's now check out radial background gradients.

Radial background gradients

It's equally simple to create a radial gradient in CSS. These typically begin from a central point and spread out smoothly in an elliptical or circular shape.

Here's the syntax for a radial background gradient (you can play with it in `example_09-04`):

```
.radial-gradient {
  margin: 1rem;
  width: 400px;
  height: 200px;
  background: radial-gradient(12rem circle at bottom, yellow, orange, red);
}
```

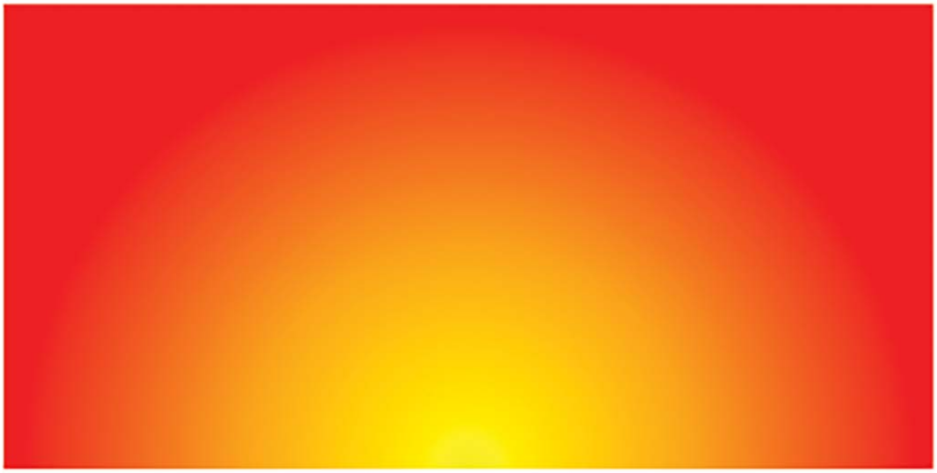


Figure 9.4: We can set a gradient to start at any point. Here, our "sunrise" gradient starts bottom center

After specifying the property (`background:`), we begin the `radial-gradient` notation. To start with, before the first comma, we define the shape or size of the gradient and the position. We used `12rem circle` for the shape and size previously, but consider some other examples:

- `circle` would be a circle, the full size of the container. The size of a radial gradient defaults to `farthest-corner` if omitted—more on sizing keywords shortly.
- `5em` would be a circle 5 em in size. It's possible to omit the `circle` part if giving just a size.
- `ellipse` would create an ellipse shape that would fit within the element.
- `40px 30px` would be an ellipse as if drawn inside a box 40px wide by 30px tall.

Next, after the size and/or shape, we define the position. The default position is `center`, but let's look at some other possibilities and how they can be defined:

- `at top right` starts the radial gradient from the top right
- `at right 100px top 20px` starts the gradient 100px from the right edge and 20px from the top edge
- `at center left` starts it halfway down the left side of the element

We end our size, shape, and position parameters with a comma and then define any color stops, which work in exactly the same manner as they do with linear gradients.

To simplify the notation, we use size, shape, and position before the first comma, then as many color stops as needed after it (with each stop separated by commas).

When it comes to sizing things such as gradients, CSS provides some keywords that are often a better choice than hardcoded values, especially with responsive designs.

Handy extent keywords for responsive sizing

For responsive work, you may find it advantageous to size gradients proportionally rather than using fixed pixel dimensions. That way, you know you are covered (both literally and figuratively) when the size of elements changes. There are some handy sizing keywords that can be applied to gradients.

You would write them like this, in place of any size value:

```
background: radial-gradient(closest-side circle at center, #333, blue);
```

Here is what each of them does:

- `closest-side`: The shape meets the side of the box nearest to the center (in the case of circles), or meets both the horizontal and vertical sides that are closest to the center (in the case of ellipses)
- `closest-corner`: The shape meets exactly the closest corner of the box from its center
- `farthest-side`: The opposite of `closest-side`, in that rather than the shape meeting the nearest side, it's sized to meet the one farthest from its center (or both the furthest vertical and horizontal side in the case of an ellipse)
- `farthest-corner`: The shape expands to the farthest corner of the box from the center
- `cover`: This is identical to `farthest-corner`
- `contain`: This is identical to `closest-side`

Practically, the two I find myself using most of the time are `cover` and `contain`. They are merely aliases for other keywords, but I find their names are self-explanatory, and they are the same keywords you can use for `background-size`. We will look at those keywords again with `background-size` later in the chapter.

You can read the W3C specification for radial background gradients at `https://www.w3.org/TR/css-images-3/#radial-gradients`.

The cheat's way to perfect CSS linear and radial gradients

If defining gradients by hand seems like hard work, there are some great online gradient generators. If you don't need the latest color formats, then `https://www.colorzilla.com/gradient-editor/` is a good one. It uses a graphics editor-style GUI, allowing you to pick your colors, stops, gradient style (linear and radial gradients are supported), and even the color space (hex, RGB(A), or HSL(A)) you'd like the final gradient in. There are also loads of preset gradients to use as starting points. Still not convinced? How about the ability to generate a CSS gradient based on the gradient values in an existing image? Thought that might swing it for you.

If it's the latest color formats you need, then take a look at `https://davidjohnstone.net/lch-lab-colour-gradient-picker`.

Conic gradients

Where radial gradients pass through any color stops on their way out from a center point, conic gradients pass through any color stops *around* a center point. We set an origin point for the gradient and then specify the color stops as angles around that center point. This accounts for the fact that these types of gradients are also known as **angle** gradients.

Here is an example conic gradient going from white to green and back to white:

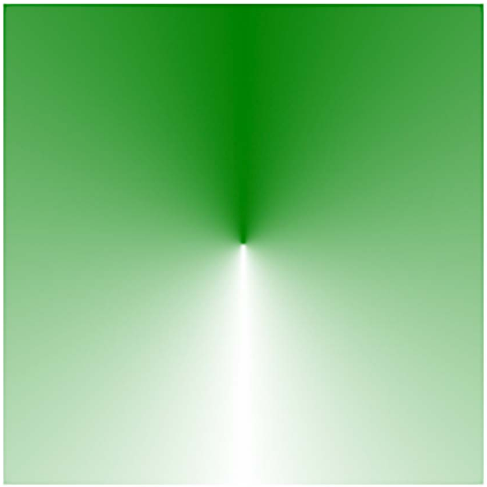


Figure 9.5: A conic gradient goes around a point, instead of outward from the point

The code to achieve that looks like this:

```
.conic-gradient {
  width: 200px;
  height: 200px;
  background: conic-gradient(at center, green, white, green);
}
```

If you have looked at the syntax for linear and radial gradients in this chapter already, the conventions will be familiar. You specify the `at` position for the gradient to start, and then the color stops along the gradient. If you omit the position of the colors, they will be split equally, so the prior example is equivalent to the following:

```
.conic-gradient {
  width: 200px;
  height: 200px;
  background: conic-gradient(at center, green 0deg, white 180deg,
    green 360deg);
}
```

The one thing that did catch me out when I first started using them is that the angle for the stops isn't just written as degrees around a circle, as you might assume; it can also be written as a percentage, like other gradients. Imagine that you wanted to change the prior example to be solid green halfway around, so it looks like this:

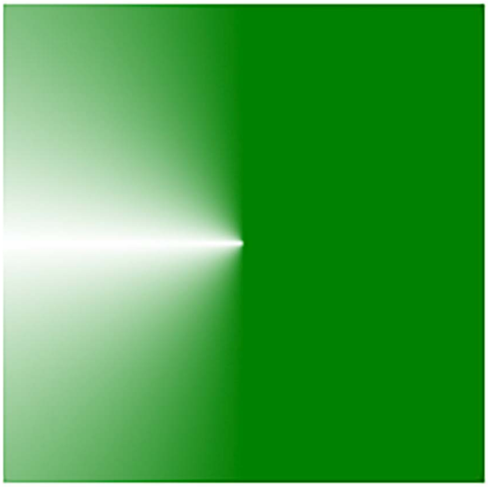


Figure 9.6: Stops around the center point are typically set in percentages

Here is the code you would use. Just adding a `50%` stop on the green or, alternatively, `180deg`, would produce an identical result:

```
.conic-gradient-with-percentage {
  width: 200px;
  height: 200px;
  background: conic-gradient(at center, green 50%, white, green);
}
```

One final trick. We can make solid sections with our conic gradients. After the first color stop, if you set another color stop and set it to start at `0deg`, it appears as a solid color. Here is an example of a single `div` element using a conic gradient to make four different color sections:

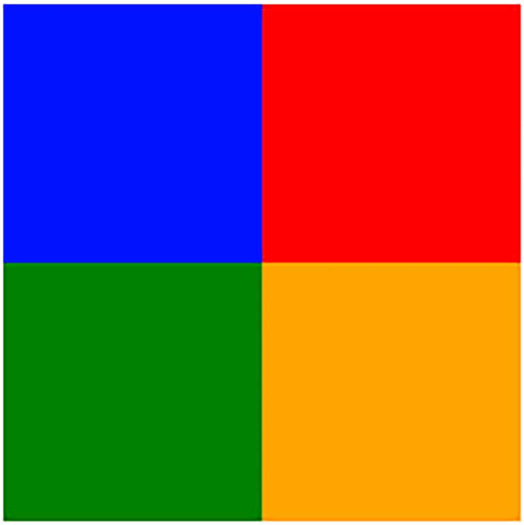


Figure 9.7: A single element using a conic gradient to create color sections

This works because the subsequent stops also start at the beginning, but by the time they are visible beneath the color that is listed before, they are already a solid color. Here is the full code for that example:

```
.conic-gradient-hard-stops {
  width: 200px;
  height: 200px;
  background: conic-gradient(at center, red 0 25%, orange 0% 180deg,
    green 0% 75%, blue 0%);
}
```

I've even mixed and matched the units in there to prove you can, but it's usually easier to stick to either percentages or angles in a single gradient—if only for the sake of your own sanity!

Now, if you were a major internet company and wanted to create a nice spinner, all you'd need would be a `border-radius` value of `50%` and a CSS animation. I'll leave that as an exercise for you, dear reader.

> You can get hard stops in linear gradients and radial gradients in the same way. Simply use a value of zero as the start of the gradient for subsequent color stops, and it will run "behind" the prior one, creating a "hard" transition from one color to the next. I've found this technique very useful when creating bar charts and similar pieces of UI. For example, imagine a bar chart with a 25/75% split. We could achieve that like this:
>
> ```
> linear-gradient(to right, goldenrod 25%, hotpink 0%)
> ```

Repeating gradients

CSS also gives us the ability to create repeating background gradients. Let's take a look at how it's done:

```
.repeating-radial-gradient {
  background: repeating-radial-gradient(black 0px, orange 5px,
    red 10px);
}
```

Here's how that looks (don't look for long, it may cause nausea):

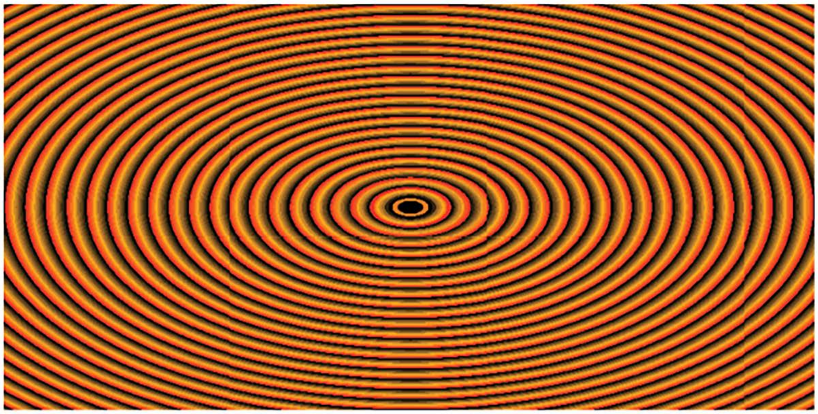


Figure 9.8: You can use repeating gradients to create all manner of visual effects

Firstly, prefix `linear-gradient` or `radial-gradient` with `repeating-`. It then follows the same syntax as a normal gradient. Here, I've used pixel distances between the black, orange, and red colors (`0px`, `5px`, and `10px`, respectively), but you could also choose to use percentages.

> You can read the W3C information on repeating gradients at `https://www.w3.org/TR/css-images-3/#repeating-gradients`.

There's one more way of using background gradients I'd like to share with you.

Background gradient patterns

Although I've often used subtle linear gradients in designs, I've found less practical use for radial, conic, and repeating gradients. However, clever folks out there have harnessed the power of gradients to create background gradient patterns. Let's look at an example from Lea Verou's collection of CSS background patterns, available at `https://projects.verou.me/css3patterns/`:

```
.carbon-fibre-background {
  margin: 1rem;
  width: 800px;
  height: 400px;
  background:
    radial-gradient(black 15%, transparent 16%) 0 0,
    radial-gradient(black 15%,.transparent 16%) 8px 8px,
    radial-gradient(rgb(255, 255, 255, 0.1) 15%, transparent 20%)
      0 1px,
    radial-gradient(rgb(255, 255, 255, 0.1) 15%, transparent 20%)
      8px 9px;
  background-color: #282828;
  background-size: 16px 16px;
}
```

Here's what that gives us in the browser—a carbon fiber background effect:

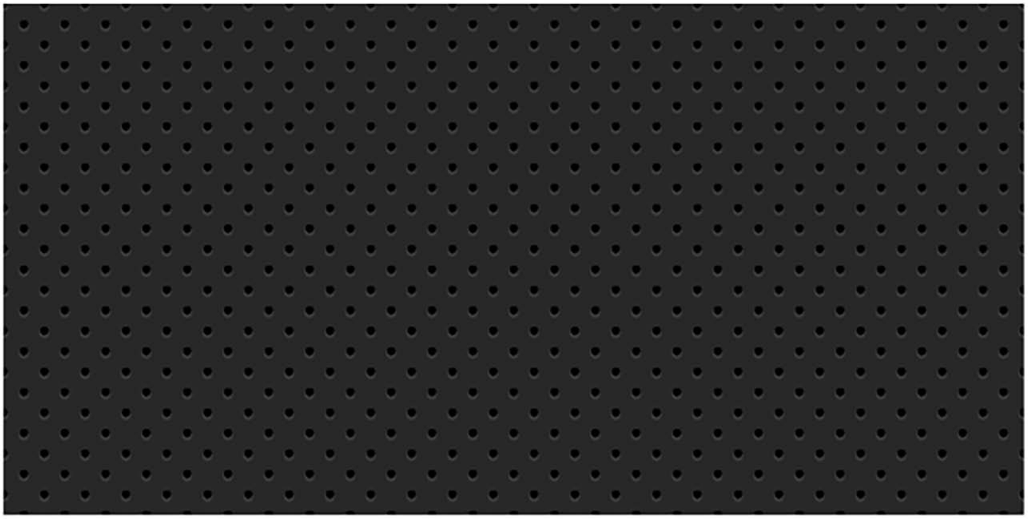


Figure 9.9: A carbon fiber effect made with pure CSS

How about that? Just a few lines of CSS and we have an easily editable, responsive, and scalable background pattern.

You might find it useful to add `background-repeat: no-repeat` at the end of the rule to better understand how it works.

As ever, thanks to media queries, different declarations can be used for different responsive scenarios. For example, although a gradient pattern might work well at smaller viewports, it might be better to go with a plain background at larger ones:

```
@media (min-width: 45rem) {
  .carbon-fibre {
    background: #333;
  }
}
```

You can view this example in `example_09-05`.

So far, we have looked at many ways of creating background images using just CSS; however, whether that is linear gradients, radial gradients, or repeating gradients, it is still just one background image. What about when you want to deal with more than one background image at the same time?

Multiple background images

Although a little out of fashion at the moment, it used to be a fairly common design requirement to build a page with a different background image at the top of the page than at the bottom, or perhaps to use different background images for the top and bottom of a content section within a page. With CSS, you can stack as many background images as you need on an element.

Here's the syntax:

```
.bg {
  background: url('../img/1.png'), url('../img/2.png'),
    url('../img/3.png');
}
```

As with the stacking order of multiple shadows, the image listed first is layered nearest to the top, or closer to the user, in the browser. You can also add a general color for the background in the same declaration if you wish, like this:

```
.bg {
  background: url('../img/1.png'), url('../img/2.png'),
    url('../img/3.png') left bottom, black;
}
```

Specify the color last, and this will show up below every image specified in the preceding code snippet.

With multiple background images, as long as your images have transparency, any partially transparent background image that sits on top of another will show through below. However, background images don't have to sit on top of one another, nor do they all have to be the same size.

Background size

To set different sizes for each image, use the `background-size` property. When multiple images have been used, the syntax works like this:

```
.bg {
  background-size: 100% 50%, 300px 400px, auto;
}
```

The `size` values (first width, then height) for each image are declared, separated by commas, in the order they are listed in the background property. As in the preceding example, you can use percentage or pixel values for each image alongside the following:

- `auto`: This sets the element at its native size
- `cover`: This expands the image, preserving its aspect ratio, to cover the area of the element
- `contain`: This expands the image to fit its longest side within the element while preserving the aspect ratio

Having considered size, let's also think about position.

Background position

If you have different background images, at different sizes, the next thing you'll want is the ability to position them differently. Thankfully, the `background-position` property facilitates that.

Let's put all this background image capability together, alongside some of the responsive units we looked at in previous chapters.

We'll create a simple space scene made with a single element and three background images, set at three different sizes, and positioned in three different ways:

```
.bg-multi {
  height: 100vh;
  width: 100vw;
  background: url('rosetta.png'), url('moon.png'), url('stars.jpg');
  background-size: 75vmax, 50vw, cover;
  background-position: top 50px right 80px, 40px 40px, top center;
  background-repeat: no-repeat;
}
```

You'll see something like this in the browser:

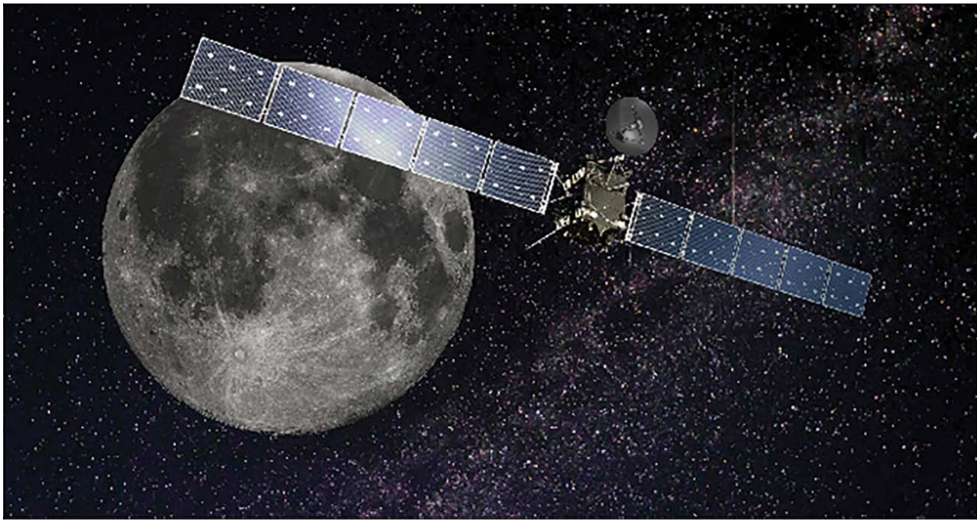


Figure 9.10: Multiple background images on a single element

We have the image of the stars at the bottom, then the moon, and finally an image of the Rosetta space probe on top. View this for yourself in `example_09-06`. Notice that if you adjust the browser window, the responsive length units work well (`vmax`, `vh`, and `vw`) and retain proportion, while pixel-based ones do not.

Where no `background-position` value is declared, the default position of top left is applied.

The background-blend-mode property

You can control how multiple background elements blend in with one another with the `background-blend-mode` property. It uses the same values as `mixed-blend-mode`, which we will look at shortly.

Background shorthand

There is a shorthand method of combining the different background properties together.

However, my experience so far has been that it produces erratic results. Therefore, I recommend the longhand method and that you declare the multiple images first, then the size, and then the position.

You can read the W3C documentation on multiple background elements here: `https://www.w3.org/TR/css-backgrounds-3/`.

High-resolution background images

Thanks to `media` queries, we have the ability to load in different background images, not just at different viewport sizes, but also at different viewport resolutions.

For example, here is the official way of specifying a background image for a "normal" and a "high" DPI screen. You can find this in `example_09-07`:

```
.bg {
  background-image: url("bg.jpg");
}
@media (min-resolution: 1.5dppx) {
  .bg {
    background-image: url("bg@1_5x.jpg");
  }
}
```

The `media` query is written exactly the same as with `width`, `height`, or any of the other capability tests. In this example, we are defining the minimum resolution that `bg@1_5x.jpg` should use as `1.5dppx` (**device pixels per CSS pixel**). We could also use `dpi` (**dots per inch**) or `dpcm` (**dots per centimeter**) units if preferred. However, despite the poorer support, I find `dppx` the easiest unit to think about; as `2dppx` is twice the resolution, `3dppx` would be three times the resolution. Thinking about that in `dpi` is trickier. "Standard" resolution would be `96dpi`, twice that resolution would be `192dpi`, and so on.

A brief note on performance

Just remember that large images can potentially slow down the feel of your site and lead to a poor experience for users. While a background image won't block the rendering of the page (you'll still see the rest of the site drawn to the page while you wait for the background image), it will add to the total weight of the page, which is important if users are paying for data.

Earlier in this chapter, I told you we would look at more shadows when we got to dealing with CSS filters. That time has come.

CSS filters

There is a glaring problem with `box-shadow`. As the name implies, it is limited to the rectangular CSS box shape of the element it is applied to. Here's a screengrab of a triangle shape made with CSS with a box shadow applied:

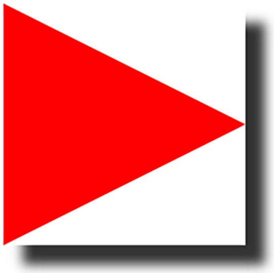


Figure 9.11: Box shadows don't always provide the effect you want

Not exactly what I was hoping for. Thankfully, we can overcome this issue with CSS filters, part of the Filter Effects Module Level 1 (`https://www.w3.org/TR/filter-effects/`).

Here is that same element with a CSS `drop-shadow` filter applied instead of `box-shadow` (you can view the code in `example_09-08`):

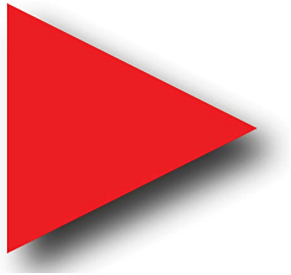


Figure 9.12: A drop-shadow filter effect can be applied to more than just boxes

Here is the format for CSS filters:

```
.filter-drop-shadow {
  filter: drop-shadow(8px 8px 6px #333);
}
```

After the `filter` property, we specify the filter we want to use, which is `drop-shadow` in this example, and then pass in the arguments for the filter. `drop-shadow` follows a similar syntax to `box-shadow`, so this one is easy; *x* and *y* offset, blur, then spread radius (both blur and spread are optional), and finally, color (also optional, although I recommend specifying a color for consistency).

CSS filters are actually based on SVG filters, which have wider support. We'll look at the SVG-based equivalent in *Chapter 11*, *SVG*.

Available CSS filters

There are a few filters to choose from. We will look at each. While images of most of the filters follow, if you are reading a copy of this book with monochrome images, you may struggle to notice the differences. If you're in that situation, remember you can still view the various filters in the browser by opening `example_09-08`. I'm going to list them out now with a suitable value specified. As you might imagine, a greater value means more of the filter is applied. Where images are used, the image is shown after the relevant code:

- `filter: url('./img/filters.svg#filterRed')`: This lets you specify an SVG filter to use.
- `filter: blur(2px)`: This uses a single length value (but not as a percentage):

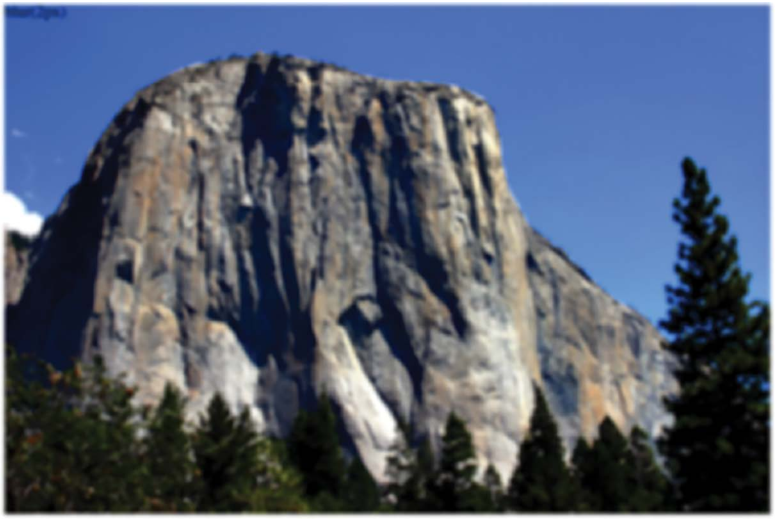


Figure 9.13: A blur filter applied

- `filter: brightness(2)`: With brightness, a value of `1` or `100%` is normal; less than that, such as `0.5` or `50%`, darkens, and more, such as `200%` or `2`, lightens:

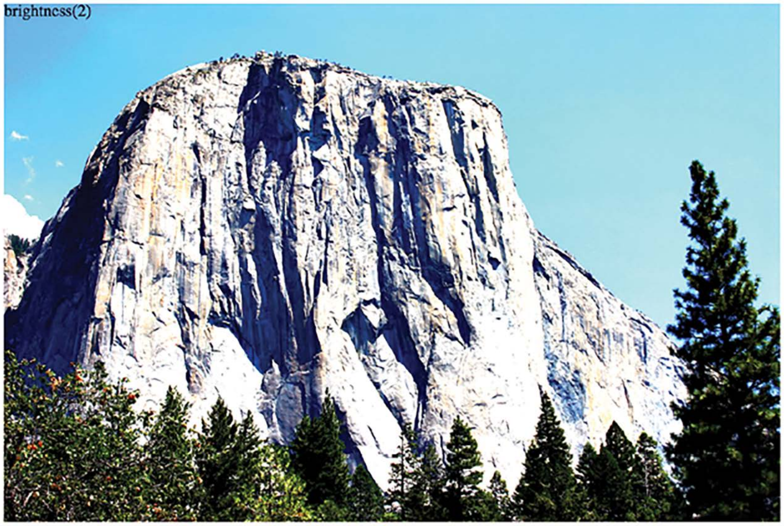


Figure 9.14: A brightness filter applied

- `filter: contrast(2)`: A value of `1` or `100%` is normal; less than that, for example, `0.5` or `50%`, reduces contrast, and more, such as `200%` or `2`, increases it:

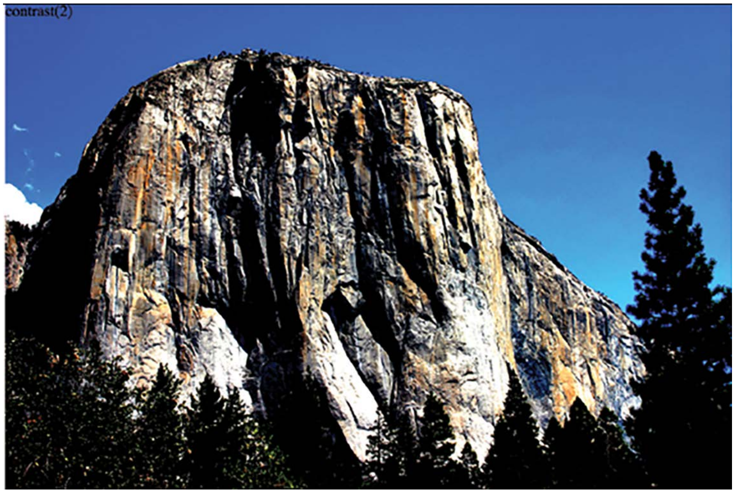


Figure 9.15: A contrast filter applied

- `filter: drop-shadow(8px 8px 6px #333)`: We looked at `drop-shadow` in detail previously.
- `filter: grayscale(0.8)`: Use a value from `0` to `1` or `0%` to `100%` to apply varying amounts

of grayscale to the element. A value of `0` would be no grayscale, while a value of `1` would be fully grayscale:

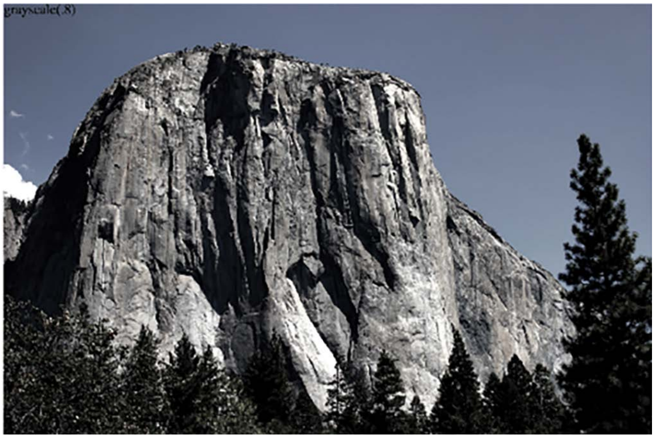


Figure 9.16: A grayscale filter applied

- `filter: hue-rotate(25deg)`: Use a value between `0` and `360` degrees to adjust the hue of the colors around the color wheel. You can use a negative value to move the "wheel" backward, and a number greater than `360` just spins it around more! Only really useful if you need to programmatically keep adding to the hue:

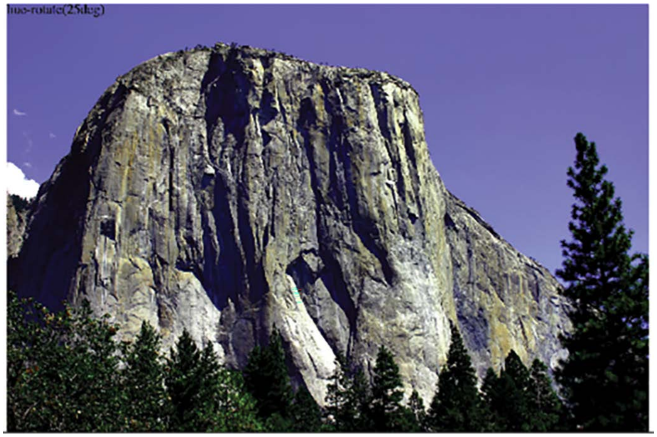


Figure 9.17: A hue-rotate filter applied

- `filter: invert(75%)`: Use a value from `0` to `1` or `0%` to `100%` to define the amount the element has its colors inverted:

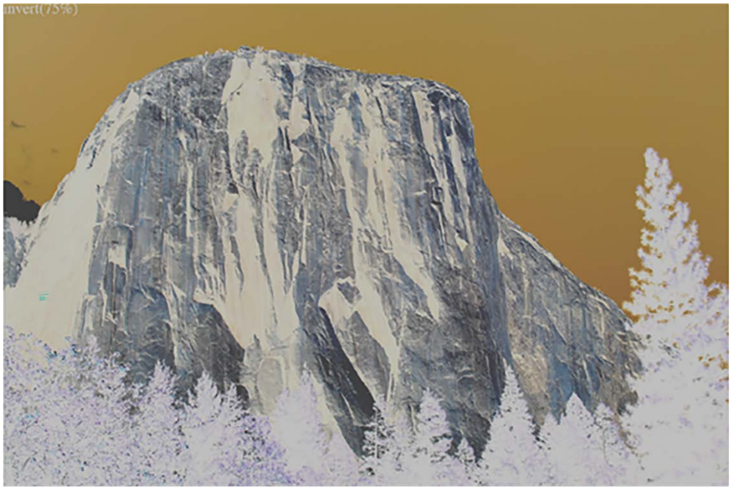


Figure 9.18: An invert filter applied

- `filter: opacity(50%)`: Use a value from `0` to `1` or `0%` to `100%` to alter the opacity of the element. `1` or `100%` is fully opaque, while `0` or `0%` would be full transparency. This is similar to the `opacity` property you will already be familiar with. However, filters, as we shall see, can be combined, and this allows opacity to be combined with other filters in one go:

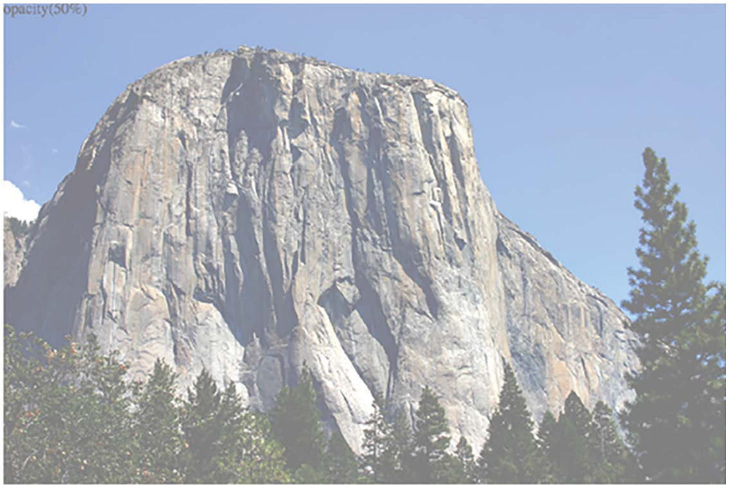


Figure 9.19: An opacity filter applied

- `filter: saturate(15%)`: Use a value from `0` to `1` or `0%` to `100%` to desaturate an image, and anything above `1/100%` to add extra saturation:

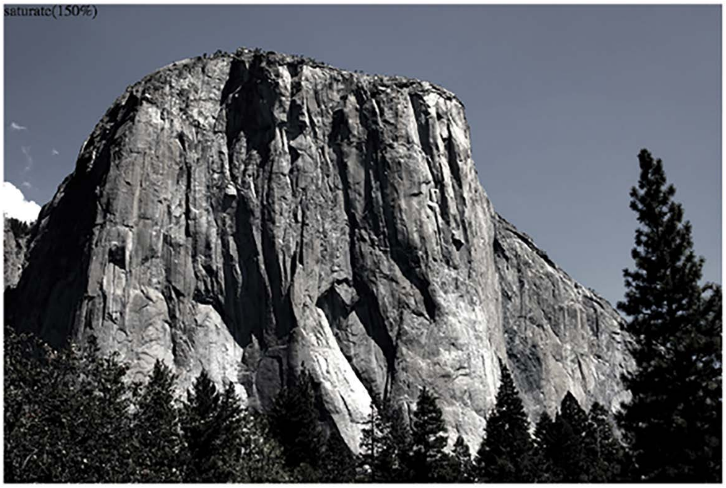


Figure 9.20: A saturate filter applied

- `filter: sepia(0.75)`: Use a value from `0` to `1` or `0%` to `100%` to make the element appear with a more sepia color. `0/0%` leaves the element as-is, while anything above that applies greater amounts of sepia, up to a maximum of `1/100%`:

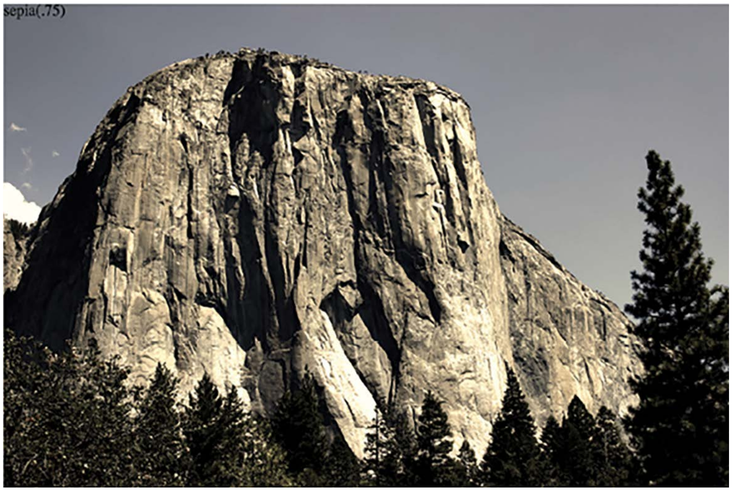


Figure 9.21: A sepia filter applied

Combining CSS filters

You can also combine filters easily; simply space-separate them. For example, here is how you would apply `opacity`, `blur`, and `sepia` filters at once:

```
.MultipleFilters {
  filter: opacity(10%) blur(2px) sepia(35%);
}
```

> Apart from `hue-rotate`, when using filters, negative values are not allowed.

I think you'll agree that CSS filters offer some pretty powerful effects. There are also effects that we can transition and transform from situation to situation. We'll look at how to do that in *Chapter 12, Transitions, Transformations, and Animations*.

> **The backdrop-filter property**
>
> There is a similar related property that can make use of all the same filters called `backdrop-filter`. Set this on any element with a partially transparent background, and it will apply the filter to the element's background.

However, before you go crazy with these new toys, we need to have a grown-up conversation about performance.

A warning on CSS performance

When it comes to CSS performance, I would like you to remember this one thing:

> *"Architecture is outside the braces, performance is inside."*
>
> *– Ben Frain*

Let me expand on my little maxim: as far as I can prove, worrying about whether a CSS selector (the part outside the curly braces) is fast or slow is pointless. I set out to prove this some years ago here: `https://benfrain.com/css-performance-revisited-selectors-bloat-expensive-styles/`, and while the post is old, the conclusion holds true today.

However, one thing that really can grind a page to a halt, CSS-wise, is *expensive properties* (the parts inside the curly braces). When we use the term *expensive* in relation to certain styles, it simply means it costs the browser a lot of overhead. It's something the browser, or perhaps more accurately, the host hardware, finds overly taxing to do.

It's possible to make a common-sense guess about what will cause the browser extra work. It's basically anything it would have to compute before it can paint things to the screen. For example, compare a standard `div` with a flat solid background against a semi-opaque image, on top of a background made up of multiple gradients, with rounded corners and `drop-shadow`. The latter is more expensive; it will result in far more computational work for the browser and, subsequently, cause more overhead.

Therefore, when you apply effects such as filters, do so judiciously and, if possible, test whether the page speed suffers on the lowest-powered devices you are hoping to support. At the least, switch on development tool features, such as continuous page repainting in Chrome, and toggle any effects you think may cause problems. This will provide you with data (in the form of a millisecond reading of how long the current viewport is taking to paint) to make a more educated decision on which effects to apply. The lower the figure, the faster the page will perform—although be aware that browsers/platforms vary, so, as ever, test on real devices where possible. Generally, if you suspect something is slowing things down, just toggling those properties off in the dev tools will show an improvement. These tools just give you hard data to make a more educated decision.

For more on this subject, I recommend the following resource: `https://web.dev/rendering-performance/`.

Compositor frame time
22.9 20.5-26.5
GPU raster: off (device)
GPU memory
26.8 MB used
512.0 MB max

Figure 9.22: Check that fancy effects aren't killing your performance

OK, grown-up talk is finished now. Let's get back to looking at the fun and powerful stuff we can do with CSS. Next on the list is clipping things.

CSS clip-path

The `clip-path` property allows you to "clip" an element with a shape. Think of clipping just like drawing a shape on a piece of paper and then cutting around it. This shape can be something simple (such as an ellipse), something more complicated (such as a polygon), or something more complex still, such as a shape defined by an inline SVG path. If you want to view each of these on a page, check out `example-08_09` in this chapter's downloadable code.

CSS basic shapes

You can use `clip-path` with any of the CSS basic shapes. These are `inset`, `circle`, `ellipse`, and `polygon`, as described here: `https://www.w3.org/TR/css-shapes-1/#supported-basic-shapes`.

Let's take a look at how we would write each of these.

clip-path with a circle

With `clip-path: circle()`, the first argument you pass is the size, and the second, which is an optional argument, is the position of that shape. So, if you wanted to clip an element down to a circle 20% of the element's height and width, you could use the following:

```
clip-path: circle(20%);
```

If you wanted the same circle `clip-mask` but positioned 60% horizontally and 40% vertically, use this instead:

```
.clip-circle {
  clip-path: circle(35% at 60% 40%);
}
```

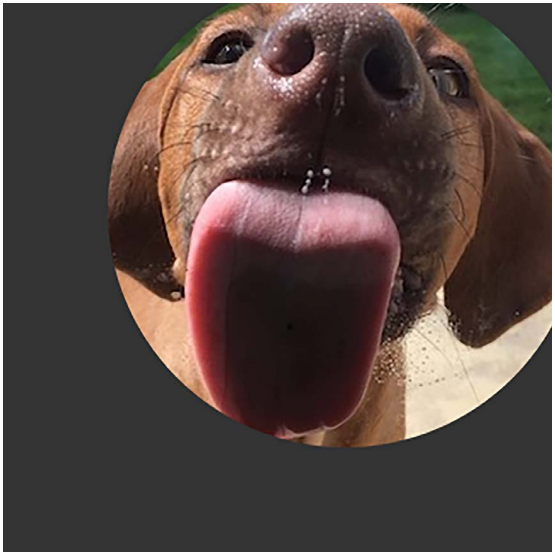


Figure 9.23: A circular clip path applied

Notice the `at` keyword in there? That's needed to communicate that the lengths after it relate to positioning and not size.

It's important to know that pointer events don't occur for areas of an element that have been clipped by a clip mask! To extend our "cutting a shape from a piece of paper" analogy, the bits outside a clip mask are like the discarded pieces of paper that have been cut away. This is useful because you won't be getting clicks on an area where you can't see the subject.

clip-path with ellipse

With `clip-path: ellipse()`, the first argument is the *x* axis radius (horizontal) and the second argument is the *y* axis radius (vertical). Just like `circle()`, you specify positioning lengths after the `at` keyword:

```
.clip-ellipse {
  clip-path: ellipse(100px 50px at 60% 40%);
}
```

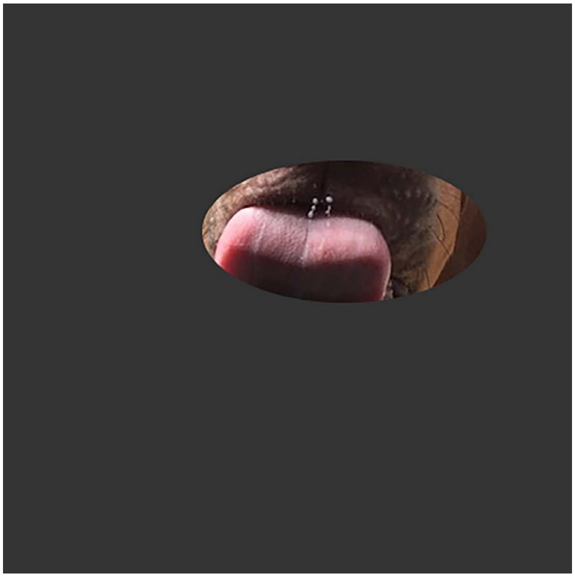


Figure 9.24: An elliptical clip path applied

clip-path with inset

The `inset()` function for `clip-path` is a little different. You pass four lengths, and these are how much you want to inset the mask from the edges. Just like when you set `margin`, the values go clockwise: top, right, bottom, and left. Just like `margin`, you can also pass just two length values; the first will be top and bottom, and the second will be left and right (the three-value syntax works the same, too).

There is also an optional `round` keyword that can be used, followed by another length; this sets how much you would like each corner to be rounded by.

Here is an example with four inset values passed and four different radius values for the corners:

```
.clip-inset {
  clip-path: inset(40px 20px 40px 20px round 0 30px 15px 40px);
}
```

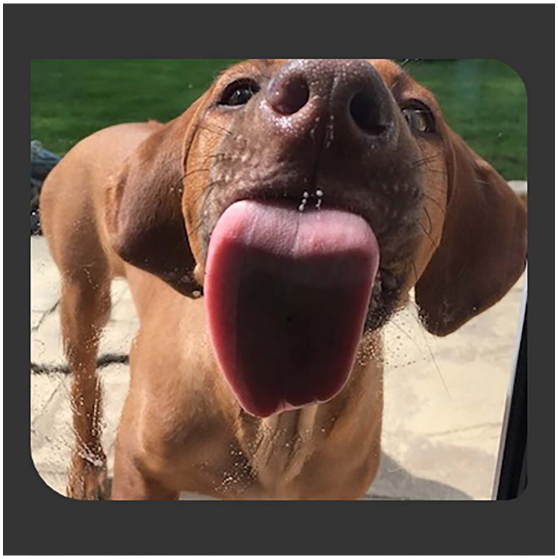


Figure 9.25: An inset clip path applied

clip-path with polygon

The `polygon()` function for `clip-path` allows us to describe a simple shape with a series of comma-separated *x* and *y* coordinates:

```
.clip-polygon {
  clip-path: polygon(50% 60px, 100% calc(100% - 40px), 0%
    calc(100% - 40px));
}
```

This is what a triangle written with the `polygon()` function looks like:

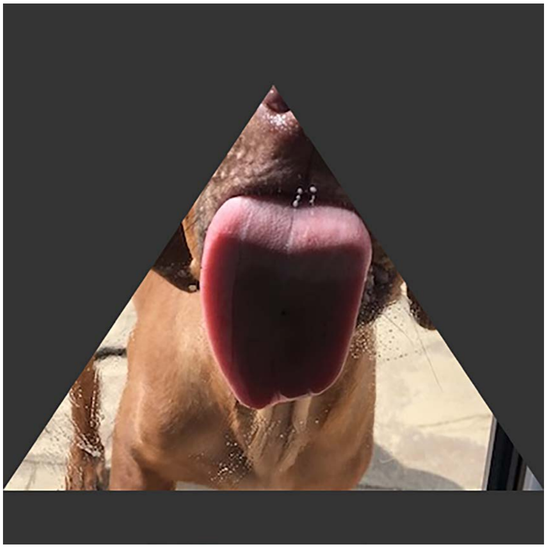


Figure 9.26: A clip path applied with a polygon

Think of this as a way of describing where to start cutting (the first argument), where to go next (any subsequent arguments), and where the last point should be. The polygon then connects the final point to the first to complete the path.

Each argument is made up of an *x* coordinate described from the top left of the containing box (for example, 50% along), and then the *y* coordinate (for example, 60px from the top) in relation to the top of the containing box. This gives you the ability to let things scale with percentages or use harder pixel values, depending on your requirements.

My favorite way to get started with a `clip-path` polygon is to use Bennett Feely's *Clippy* website: `https://bennettfeely.com/clippy/`.

There are a bunch of premade shapes to choose from, as well as a GUI for amending the shape. Highly recommended!

clip-path with URL (clipping source)

You can also pass `clip-path` a **clipping source** via a URL. The URL needs to be an SVG `clipPath` somewhere in the document.

I made a star shape as an SVG path and added an `id` value of `starSymbol` to the `clipPath` element. With that in place, you can tell `clip-path` to use that path like this:

```
.clip-url {
  clip-path: url(#starSymbol);
}
```

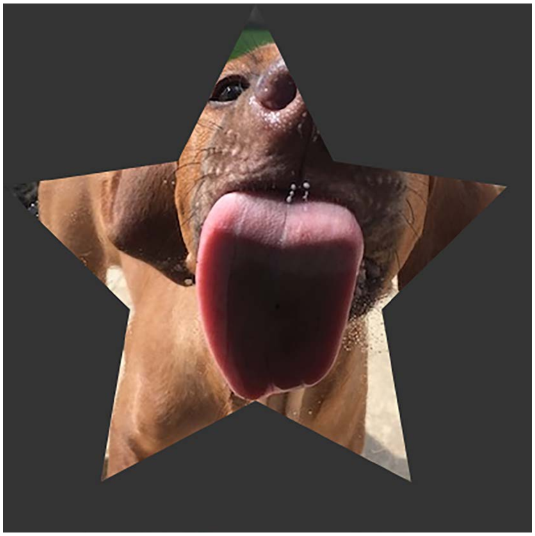


Figure 9.27: A clip path applied with an SVG path

You can view each of these `clip-path` examples in `example_09-09`.

The shape() CSS function

In 2025, most browsers have added support for the CSS `shape()` function. This function is a way of expressing all that you can do with an SVG path, in CSS, with some extra niceties to boot. We cover SVG in great detail in *Chapter 11, SVG*.

The `shape()` function is a pretty sophisticated and fully featured function, and as such, we won't be covering all that it is capable of here. Instead, I'll share an overview of the capabilities and some examples, relate where I think it is most useful, and point you toward some excellent resources, where developers far more skilled than I with `shape()` can explain its possibilities in greater detail.

The essential information is that `shape()` allows you to describe complex shapes using commands such as `line`, `arc`, `curve`, and such. You can use `shape()` with `clip-path`, but also `offset-path`, which allows an element to follow a complex shape with an animation.

I'll be honest, as I'm not great with mathematics, I generally "cheat" with the `shape()` function. By *cheat*, I mean draw the shape I want in an SVG editing tool such as Figma or Sketch, export that shape as an SVG, take the resultant path from the SVG (the part inside the `d=""` attribute), and then paste that into an SVG-to-CSS conversion tool: `https://css-generators.com/svg-to-css/`. That provides the output for the `shape()` function to display it. Let's do that now, together, and you can find the source SVGs in `example_09-10`.

First of all, here is my first shape I have as an SVG:

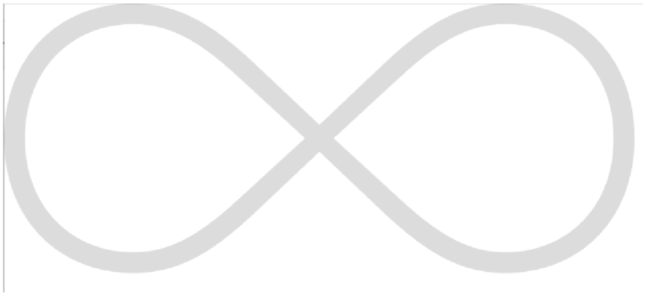


Figure 9.28: The SVG we will steal the path from

And that looks like this as code. Be aware that I have truncated the values in the `d=""` attribute for brevity:

```
<?xml version="1.0" encoding="UTF-8"?>
<svg width="546px" height="234px" viewBox="0 0 546 234" version="1.1"
    xmlns="http://www.w3.org/2000/svg"
        xmlns:xlink="http://www.w3.org/1999/xlink">
    <title>Shape</title>
    <g id="Page-1" stroke="none" stroke-width="1"
        fill="none" fill-rule="evenodd">
        <g id="Group" fill="#DCDCDC">
            <path d="M0,116.588028 C0,186.821278 47.0555108,233.175125
                111.83782,233.175125 C151.212999,233.175125 180.914908...
                    Z" id="Shape" fill-rule="nonzero"></path>
        </g>
    </g>
</svg>
```

Now, I'm going to copy out the contents of that `d=""` attribute, which begins `M0,116`, and ends in `Z`.

Then, head over to `https://css-generators.com/svg-to-css/` and paste that content into the input box, and you will see some output similar to this. Again, be aware that the content of the `shape()` function is truncated here for brevity, but you can see the full content in the example code:

```
.shape {
  aspect-ratio: 2.34;
  clip-path: shape(from 0% 50%,
    curve to 20.5% 100% with 0% 80.12%/8.62% 100%,
      curve to 39.83% 77.47% with 27.72% 100%/33.16% 92.21%,…close);
}
```

Now, if you have opened the `index.html` file inside `example_09-10`, you will see something especially useful about using `shape()` for shapes. That is, we can animate the background with gradients with the help of the CSS `@property` at rule, which we will look at in more detail in *Chapter 12, Transitions, Transformations, and Animations*. In this instance, it's a hexagon shape, with an animating conic gradient. I encourage you to take a look for yourself in the browser.

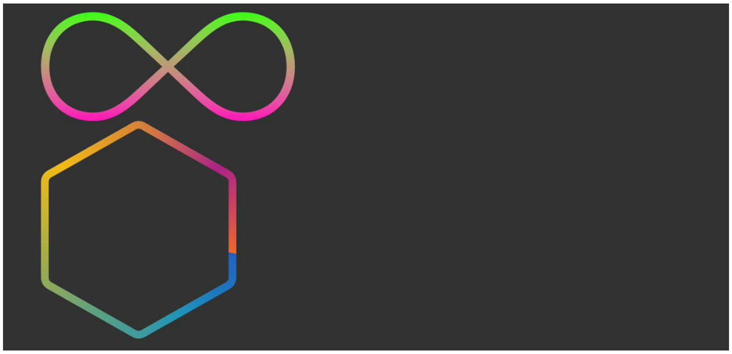


Figure 9.29: You can animate the background of shape() clip paths

More resources exploring shape()

The best examples and explanations on using `shape()` I have found are by Temani Afif. Head on over to his site at `https://css-articles.com`, where he has links out to his many articles on creating all manner of shapes with extremely thorough examples. If you have a need to go deep on the `shape()` function, I can recommend no better guide.

Animating clip-path

As if all this clipping malarkey wasn't interesting enough, you can also animate a `clip-path`, as long as there remains the same number of points in the shape. For example, you can animate a triangle into a different-shaped triangle, but without some tricks, you can't convert that same triangle into a star. I say "without some tricks" because you can effectively do this by hiding a few points along the lines of the polygon so your triangle has enough points to animate into a star.

Here's a screengrab of an extra example I added, `example_09-11`. This is just to hopefully give you an idea of what can be done by adding a few of the things we have already learned and a few things we will learn in future chapters together:

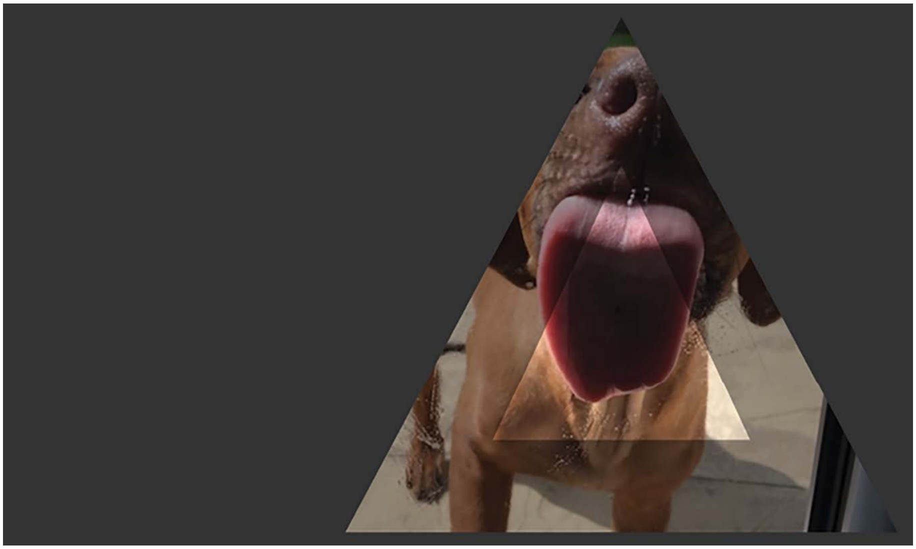


Figure 9.30: You can even animate clip-path

This shows two elements with the same background image being masked with `clip-path`. You can also tint the image with an overlay like this:

```
background: linear-gradient(hsla(0, 0%, 0%, 0.4) 0, hsla(0, 0%, 0%, 0.4)
  100%),
  url("image.jpg");
```

You may be wondering, if we wanted to tint the image with a color, why don't we just use a background color in the declaration?

The answer is that it doesn't work. When you set multiple backgrounds with the `background` property, it's not possible to use `background-color`, even a semi-transparent one, above an image, and have the color let the image show through beneath. However, one proven trick is to create a background image using `linear-gradient` with the same semi-opaque color at either end of the gradient, which creates the effect we are after.

We are then moving the main element around the page with one set of `keyframes` animations (we will get to animations in *Chapter 12*, *Transitions, Transformations, and Animations*) and animate the `clip-path` with another set.

I think once you start playing about with these techniques, you're going to surprise yourself with just what you can create with relative ease!

There may be occasions when you don't just want to clip what's in an element, and you actually want to overlay it with a mask. We can do that with CSS too! Let me show you.

mask-image

You can also mask elements with images, from either an image source with transparency, such as a PNG graphic, a linear gradient, which we looked at earlier in this chapter, or an SVG `mask` element. You can read about all the possibilities afforded to us in the specification here: `https://www.w3.org/TR/css-masking-1/`.

In the meantime, we will just look at a fairly straightforward example so you can appreciate the kind of effect that is possible and how the syntax works to achieve it.

Suppose we have an image. I have one that NASA took of Mars. I'd get one I took myself, but, you know, it's a bit of a jaunt.

Now, suppose we also have a PNG image that is transparent except for the word *MARS*. We can use this PNG as a mask on top of our image element.

This is what we see in the browser:

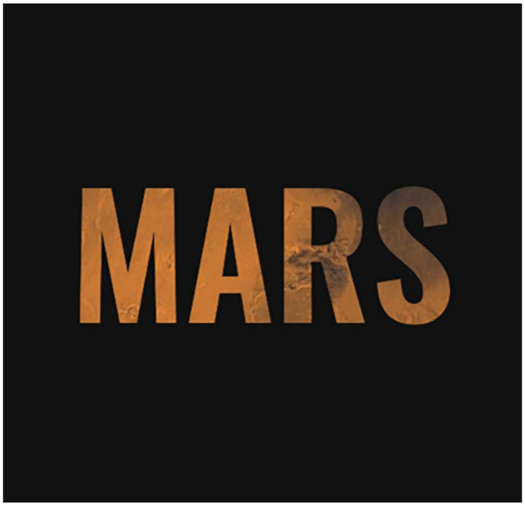


Figure 9.31: A mask image applied

Here is our relevant HTML:

```
<img
  src="mars.jpg"
  alt="An image of Mars from space"
  class="mask-image-example"
/>
```

And here is our CSS:

```
.mask-image-example {
  display: block;
  height: 900px;
  width: 1000px;
  margin: 0 auto;
  mask-image: url("mars-text-mask.png");
}
```

The only relevant part of the mask is the `mask-image` property, which tells the browser what we want to use as a mask on this element.

Now, the cynical among you might suggest the same effect could have just been achieved in a graphics package. I'd say you were right, but then raise you this:

```
.mask-image-example {
  display: block;
  height: 1024px;
  width: 1024px;
  margin: 0 auto;
  mask-image: url('mars-text-mask.png');
  animation: moveMask 6s infinite alternate;
}
@keyframes moveMask {
  0% {
    object-position: 0px 0px;
  }
  100% {
    object-position: 100px 100px;
  }
}
```

We will look at animations in *Chapter 12, Transitions, Transformations, and Animations*, but what we have done here is add an animation that moves the background image behind the mask so that the word *MARS* stays in position while the planet moves behind. Try doing that in Photoshop!

If we wanted to swap things around and have the mask move and the image of the planet stay put, it is as simple as swapping `object-position` in the animation for `mask-position`.

You can have a play with this example in `example_09-12`.

Be aware that, to see this example and the next, you may need to view the file from a server. If you have Node.js installed, you can run `npx serve` in the Terminal from the root of each example folder to install and run a local server there.

There are quite a few related properties for `mask-image` that can alter the way the mask works. If you find yourself tasked with something mask-related, be sure to look through the specifications and see whether there is anything in there that might aid your specific requirements.

Well, I don't know about you, but I'm almost exhausted with the visual CSS "cardio" we've just been through. Before you reach for the towel and water bottle, just let me tell you about `mix-blend-mode`.

mix-blend-mode

One final, very visual, property I want to relate before we end this chapter is `mix-blend-mode`. This property lets you decide how you want one element to "blend" with the element it sits on top of.

Here are the possible blend modes: `normal`, `multiply`, `screen`, `overlay`, `darken`, `lighten`, `color-dodge`, `color-burn`, `hard-light`, `soft-light`, `difference`, `exclusion`, `hue`, `saturation`, `color`, and `luminosity`.

Static images don't really do this property justice, so I'd encourage you to open `example_09-13` in a browser. We're using that same image of Mars from the last example, but setting it as a fixed background for the `body` element.

Then, we add some text on top that you can scroll to see `mix-blend-mode` in effect. In this example, we have used `overlay`, for no reason other than I thought it worked best. If you open the example along with your developer tools, you can cycle through the other possibilities. It's perhaps redundant to say so, but the options themselves will have different effects, depending on the foreground and background they are being applied to.

The specification for `mix-blend-mode` is the *Compositing and Blending Level 1* specification: `https://www.w3.org/TR/compositing-1/#mix-blend-mode`.

Summary

In this chapter, we looked at a selection of the most useful CSS features for creating lightweight aesthetics in responsive web designs. CSS's background gradients curb our reliance on images for background effects.

We considered how they can be used to create infinitely repeating background patterns. We also learned how to use text shadows to create simple text enhancements and box shadows to add shadows to the outside and inside of elements.

We also looked at CSS filters. These allow us to achieve even more impressive visual effects with CSS alone and can be combined for truly impressive results.

In the last part of this chapter, we looked at creating masking effects with both images and clipping paths, and took a look at the new `shape()` function that lets us express complex paths that we had previously required SVGs for.

Much of the wow factor with CSS is figuring out interesting ways of combining the various capabilities, simply for a desirable effect or perhaps solving a hitherto impossible task. I'm always discovering new approaches; sometimes I stumble upon them myself, more often I see some technique that someone else has developed. Either way, when I learn something, I tend to document it on my blog, so stop by there from time to time for interesting tidbits: `https://benfrain.com`.

We've added some considerable capabilities to our tool belts in this chapter! We will add more aesthetic CSS effects in *Chapter 12*, *Transitions, Transformations, and Animations*. But, before that...

In the next chapter, we're going to turn our attention to responsive images. If you work with image-heavy projects, you'll want to pay close attention. Using the right image for the right job can save users time and money. And using the right syntax to deliver the right image makes sure they get it!

10

Responsive Images

We should always aim to build websites and applications to be as lean as possible. The benefits of faster sites are well documented. As far back as 2009, Google was publishing its findings on the importance of page speed: `https://services.google.com/fh/files/blogs/google_delayexp.pdf`.

Furthermore, research carried out by Google in 2017 suggested that "*the probability of bounce increases 32% as page load time goes from 1 second to 3 seconds*" and that "*…53% of visits are abandoned if a mobile site takes longer than three seconds to load.*"

> Read more from that research here: `https://www.thinkwithgoogle.com/intl/en-gb/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/`.

Since then, there has been research by Vodafone (`https://web.dev/vodafone/`), eBay (`https://innovation.ebayinc.com/stories/speed-by-a-thousand-cuts/`), Yelp (`https://engineeringblog.yelp.com/2021/01/boosting-user-conversion-with-ux-performance-wins.html`), and many more that all come to the same conclusion.

Speed matters.

Images are a significant contributor to website **weight**, and therefore web page speed, or lack thereof. If you head over to `https://httparchive.org/reports/page-weight`, you can see, by whatever lens you choose, that page weight is made up, in large part, of images. In the period from June 2024 to June 2025, the average page weight was 2,799 KB, and that was made up of video at 3,170 KB, JavaScript at 1,013 KB, and images at 789 KB (CSS was 66 KB, fonts 142 KB, and HTML just 48 KB).

As such, it is incumbent upon us to do everything in our power to make our images as lean and relevant for the screen viewing them as we can. This, in essence, is the aim of responsive images.

For the purpose of this chapter, let us make a distinction between the two different but related goals: how to serve the *leanest* image we can to our user, and how to serve the *most relevant* image to our user. While these two ideals overlap and sometimes share the same technical solution, the distinction may make digesting the ideas simpler.

In this chapter, we will be covering the following:

- Why we should use modern image formats where possible
- The AVIF image format
- The WebP image format
- Image format comparisons
- How to progressively enhance modern image formats
- Understanding the responsive images problem space
- Using `srcset` for resolution switching
- How to achieve art direction with `picture`
- Practical considerations for responsive images

Let's begin by looking at image formats and how different formats alone can significantly reduce the amount of data our website needs to send down the wire.

Modern image formats

Historically, it was simple to choose the correct image format to use in a given scenario.

If the image needed to be animated and for some reason a video wasn't a viable option, you would opt for an animated GIF. If it was a photo, JPG was the logical choice, and if it was anything that needed transparency, it would be PNG, which, unlike JPG, supports alpha transparency.

However, if you want to save yourself and your users' bandwidth, it is no longer so simple. Let's consider two newer image formats and where they might be applicable and advantageous to use.

AVIF and WebP are formats that, given the same image as a starting point, almost always produce a smaller file size and better quality than an equivalent JPG file.

AVIF image format

AVIF, the image counterpart of AV1 video, is an open source image format for still and, crucially, animated images. It supports an alpha channel, which JPG doesn't, and compresses data far more effectively than JPEG or even WebP (which we will look at shortly).

It also supports greater color depth, allowing the use of wide gamut colors in images (you'll want to look at *Chapter 8*, *CSS Color*, for more information on color gamut). On top of that, it supports animations and "live" photos with image sequences.

If that wasn't enough, it almost always, to my eyes at least, produces a better-looking image when compared with, say, a JPG. So what's the catch? There are two: firstly, the ease with which you can create an AVIF image, and secondly, to a far lesser extent, the support in browsers for actually viewing them.

To create AVIF images, you typically need a plugin for your graphic editor of choice; few graphic programs natively support AVIF currently. As I write this, in the second half of 2025, neither Figma nor Sketch does. If you want a straightforward browser-based tool to do a conversion, look at Squoosh: `https://squoosh.app/`.

When it comes to viewing AVIF images in the browser, it wasn't until 2024 that all the browsers supported it. So if you need to service users on older browsers, this is certainly a consideration. If you are lucky enough not to have to worry about this, it's all good.

Let's also consider WebP.

WebP image format

WebP was a format created by Google. Roughly speaking, a WebP image will tend to give you a 25% file saving against a comparable PNG or JPG.

It also supports alpha and animation, so it can be a good replacement for GIF.

Making and viewing WebP is a little easier than AVIF, if only because it has been around for longer and, as such, enjoys wider support. All modern operating systems and browsers now support WebP. Even Mac-only graphics applications such as Sketch allow a native export to WebP.

While not offering quite the file savings of AVIF, WebP is a solid second choice.

There are other modern image formats, such as HEIC and JPEG XL, but both of them lack cross-browser support, so I've discounted them from our comparisons.

Image format comparisons

Let's consider an example to illustrate the possible savings. In the code that accompanies this chapter, you will find a file called `scones.jpg`. This is the same picture of scones we used in *Chapter 1, The Essentials of Responsive Web Design*, completely unmodified.

On my computer, it tells me it has a file size of 199 KB. If I load that file into the aforementioned Squoosh and leave the defaults, it tells me it can optimize the file down to a JPG of only 122 KB. That file is in the code as `squoosh-scones.jpg`. That 39% saving in itself should be enough to show how much unneeded weight our images contain.

How does image optimization work?

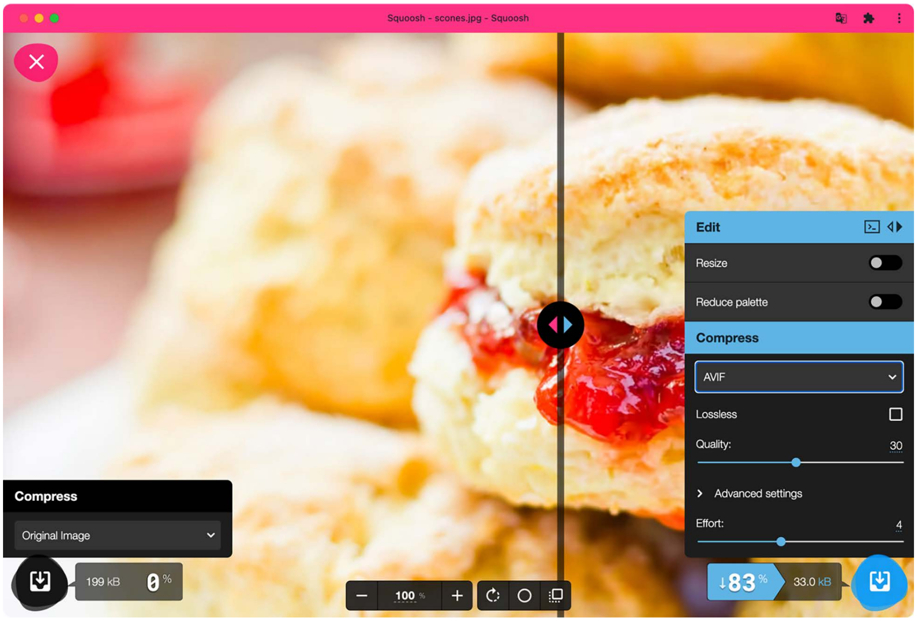
By default, an image from an application such as Photoshop or straight out of a camera contains heaps of extra information we just don't need when destined for the web: things such as the location the image was taken, the date and time, the camera used, aperture and lens settings, focal depth, and heaps, heaps more. Plus, the image can almost always have its data rewritten in a more optimized manner. That by itself would be the general idea of **lossless** optimization. At this point, we would be taking no detail from the image. However, we can often go further, removing image data with no reduction in perceived quality. That process is known as **lossy** optimization.

The trick is to find the "sweet spot" where you are saving significant extra data but not diminishing the perceptual quality of the image. At the very least, for any serious work, you should be trying to automate lossless compression of images, where no human analysis of the output will be necessary.

Let's save that same file as a WebP file; this gives us a file of only 79.4 KB, a 60% saving on the original file size. You can find that file as `squoosh-scones.webp`.

Lastly, let's save it as an AVIF file, as shown in *Figure 10.1*. By default, Squoosh sets the quality slider on AVIF to only `30`, but if I move the comparison line across onto the lovely strawberry jam of that image (gosh, this is making me hungry), we can see that some of the detail is being lost.

But even if I move it up to a quality of `40`, which I feel gives an acceptable visual, it still creates a file that is only 52.1 KB, a 74% improvement on the original! You can find that file in the code as `squoosh-scones.avif`.


Squoosh - scones.jpg - Squoosh

Edit

Resize

Reduce palette

Compress

AVIF

Lossless ☐

Quality: 30

Advanced settings

Effort: 4

↓83 % 33.0 kB

Compress

Original Image

199 kB 0 %

100 %

Figure 10.1: The Squoosh app lets you see the effect of your changes in real time

With our example file, with a few seconds' work, we were able to create a version of the image that was almost a quarter of the original size. This is but one image and one set of choices, but it should prove to you just how much unneeded data we are often serving up to our users.

And that brings us to how we actually serve up different file versions of an image. The answer is that we can use the `picture` element.

Using the picture element to progressively enhance images

We will soon see how to use the `picture` element for "art direction," serving up different images for different situations. However, I have actually found the greatest practical utility of the `picture` element is for providing alternative image formats. Let's look at that first.

The most common use case I find myself currently needing is providing an AVIF version of an image, if possible, then a WebP version for users if they can't make use of AVIF, and then a JPG if a user can't make use of an AVIF or a WebP. Here is how we can do that with `picture`:

```
<picture>
    <source srcset="scone.avif" type="image/avif" />
    <source srcset="scone.webp" type="image/webp" />
    <img
        src="scone.jpg"
        alt="A delicious scone, baked to perfection"
        loading="lazy"
    />
</picture>
```

The essence of the `picture` element is wrapping a number of images up, with the preferred options first, and the browser chooses the first one it is able to make use of. If you have already read *Chapter 2*, *Writing HTML Markup*, you will recognize the `source` element from the *Providing alternate media sources* section. The principle is identical: we specify different versions of a piece of media, and the browser chooses the first one it can make use of.

The CSS equivalent to this when using background images would be the `image-set()` function, which allows us to provide a number of possible images for the browser to chose from. If I wanted to provide a `avif`, `webp`, and `jpg` version of a file I could do it like this:

```
background-image: image-set(
    url("image.avif") type("image/avif"),
    url("image.webp") type("image/webp"),
    url("image2.jpg") type("image/jpeg"))
```

It also allows you to pass an optional 'resolution' as 1x, or 2x etc. If you do this, each image in the set must have a different resolution.

In the preceding example code, we would prefer the browser to use the `scone.avif` image, but if it can't use that, it uses `scone.webp`. If even that cannot be understood, it falls back to `scone.jpg`. Crucially, in this situation, where all the images are of the same thing, you don't need to write the `alt` text for the image more than once. Whichever image the browser chooses, it will only download that one image.

In addition, back in *Chapter 2*, we covered the `loading` attribute and noted that if we were happy for images to download in their own time, we could add `loading="lazy"` to them. The same is true here, but like the `alt` tag, you can see in our example code that we only have to specify it on the `img` tag; the two `source` elements in `picture` are merely alternate `src` values for `img`.

So, that is how we can provide modern images for browsers that support them, and older, more broadly supported formats for those that don't, all while providing solid accessibility and meaning with the `alt` tag in case the image doesn't load or the user can't make use of the visuals.

In the prior code, we are just dealing with a single image and different formats of it. That addresses our problem of file weight. Now, let's look at how to deal with serving the best image given the situation. For that, we will put aside `picture` momentarily and make use of another tool, `srcset`. But first, let's be certain we understand the problem we are trying to solve.

The inherent problem of responsive images

As an author, you cannot know about every possible device that may visit your site now or in the future. Only a browser knows the particulars of the device viewing a website, such as its screen size and device capabilities.

Conversely, only the people making the website know what versions of an image we have at our disposal. We may have three versions of the same image: small, medium, and large, each with increasing dimensions to cater for the anticipated screen size and screen density eventualities. The browser does not know this. We have to tell it.

To summarize the conundrum, we, as the website authors, have only half of the solution, in that we know what images we have. The browser has the other half of the solution, in that it knows what device is visiting the site and what the most appropriate image dimensions and resolution would be.

How can we tell the browser what images we have at our disposal so that it may choose the most appropriate one for the user?

In the first few years of responsive web design, there was no specified way. Thankfully, we now have the *Embedded content* specification: `https://html.spec.whatwg.org/multipage/embedded-content.html`.

The *Embedded content* specification describes ways to deal with the simple resolution switching of images, to facilitate a user on a higher resolution screen receiving a higher resolution version of images.

It also facilitates "art direction" situations for when authors want users to see a totally different image depending on a number of device characteristics (think media queries), such as a close-up image of something on smaller viewports and then a wide-angle image of the same thing for larger viewports.

Demonstrating responsive image examples is tricky. It's not possible to appreciate on a single screen the different images that could be loaded with a particular syntax or technique. Therefore, just as when we looked at serving up different file versions of an image, the examples that follow will be mainly code, and you'll just have to trust me that it's going to produce the result you need in supporting browsers. Let's look at two further scenarios that you're likely to need responsive images for. These are switching one image for another when a different resolution is needed, and changing an image entirely depending on the available viewport space.

Simple resolution switching with srcset

Let's suppose you have three versions of the same image. One is suitable for standard pixel density displays, commonly referred to as `1x`. Another image is higher resolution, to cater for higher density, or `1.5x` displays, and, finally, the third is a high `2x` resolution version for very high-density displays. Here is how we can let the browser know that we have these three versions available:

```
<img
    src="scones_small.jpg"
    srcset="scones_medium.jpg 1.5x, scones_large.jpg 2x"
    alt="a delicious looking baked scone"
/>
```

This is about as simple as things get with responsive images, so let's ensure that the syntax makes perfect sense.

First of all, the `src` attribute, which you will already be familiar with, has a dual role here; it specifies the small `1x` version of the image, and it also acts as a fallback image if the browser doesn't support the `srcset` attribute. That's why we are using it for the small image. This way, older browsers that will ignore the `srcset` information will get the smallest and best-performing image possible.

For browsers that understand `srcset`, with that attribute, we provide a comma-separated list of images that the browser can choose from. After the image name (such as `scones_medium.jpg`), we issue a simple hint.

I've specifically called it a hint rather than an instruction or a command, and you will see why in a moment. In this example, `1.5x` and `2x` have been used, but any integer would be valid. For example, `3x` or `4x` would work too (providing you can find a suitably high-resolution screen).

However, there is an issue here; a device with a `1440px` wide, `1x` screen will get the same image as a `480px` wide, `3x` screen. That may or may not be the desired effect.

Advanced switching with srcset and sizes

Let's consider another situation. In a responsive web design, it wouldn't be uncommon for an image to be the full viewport width on smaller viewports, but only half the width of the viewport at larger sizes. The main example in *Chapter 1, The Essentials of Responsive Web Design*, was a typical illustration of this. Here's how we can communicate these intentions to the browser:

```
<img
    srcset="scones-small.jpg 450w, scones-medium.jpg 900w"
    sizes="(min-width: 280px) 100vw, (min-width: 640px) 50vw"
    src="scones-small.jpg"
    alt="Lots of delicious scones"
/>
```

Inside the `img` tag, we are utilizing `srcset` again. However, this time, after specifying the images, we are adding a value with a `w` suffix. This tells the browser how wide the image is. In our example, we have a `450px` wide image (called `scones-small.jpg`) and a `900px` wide image (called `scones-medium.jpg`). It's important to note that this `w`-suffixed value isn't a "real" size. It's merely an indication to the browser, roughly equivalent to the width in "CSS pixels."

> What exactly defines a pixel in CSS? I wondered that myself. Then, I found the explanation at `http://www.w3.org/TR/css3-values/#reference-pixel` and wished I hadn't wondered.

This `w`-suffixed value makes more sense when we factor in the additional `sizes` attribute. The `sizes` attribute allows us to communicate the intentions for our images to the browser. In the preceding example, the first value is equivalent to "for devices that are at least `280px` wide, I intend the image to be around `100vw` wide."

If some of the units used, such as `vh` (where `1vh` is equal to 1% of the viewport height) and `vw` (where `1vw` is equal to 1% of the viewport width), don't make sense, be sure to read *Chapter 6, CSS Nesting, Layers, Selectors, and More*.

The second part is, effectively, "Hi browser, for devices that are at least `640px` wide, I only intend the image to be shown at `50vw`." That may seem a little redundant until you factor in **dots per inch** (**DPI**) (or **DPR** for **device pixel ratio**). For example, on a `320px` wide device with a `2x` resolution (effectively requiring a `640px` wide image, if shown at full width), the browser might decide the `900px` wide image is actually a better match as it's the first option it has for an image that would be big enough to fulfill the required size.

Did you say the browser "might" pick one image over another?

An important thing to remember is that the values given in the `sizes` attribute are merely hints to the browser. That doesn't necessarily ensure that the browser will always obey. This is a good thing.

Trust me, it really is. It means that, in the future, if there is a reliable way for browsers to ascertain network conditions, it may choose to serve one image over another because it knows things at that point that we can't possibly know at this point as the author. Perhaps a user has a setting on their device to "only download `1x` images" or "only download `2x` images." In these scenarios, the browser can make the best call.

The alternative to the browser deciding is to use the `picture` element. Using this element ensures that the browser serves up the exact image you asked for. We looked at this element earlier in the chapter to solve the problem of serving the best image format for the user's needs. This time we are going to use it to serve different images in different scenarios. Let's take a look at how that works.

Art direction with the picture element

The final scenario you may find yourself in is one in which you have different images that are applicable at different viewport sizes. For example, consider our cake-based example again from *Chapter 1*. Maybe on the smallest screens we would like a close-up of the scone with a generous helping of jam and cream on top. For larger screens, perhaps we have a wider image we would like to use. Perhaps it's a wide shot of a table loaded up with all manner of cakes. Finally, for larger viewports still, perhaps we want to see the exterior of a cake shop on a village street with people sitting outside eating cakes and drinking tea (I know, sounds like nirvana, right?).

We need three different images that are most appropriate at different viewport ranges. Here is how we could solve this with `picture`:

```
<picture>
    <source media="(min-width: 960px)" srcset="cake-shop.webp" />
    <source media="(min-width: 480px)" srcset="cake-table.webp" />
    <img src="scones.webp" alt="Lots of cakes" />
</picture>
```

Be aware that when you use the `picture` element, it is merely a wrapper to facilitate other images making their way to the `img` tag within. If you want to style the images in any way, it's the `img` tag that should get your attention. Similarly, you can't change the `alt` tag contents by depending upon which image gets loaded, so make sure you provide text that covers all eventualities as best you can.

Secondly, the `srcset` attribute here works exactly the same as in the previous example.

Thirdly, the `img` tag provides your fallback image and also the image that will be displayed if a browser understands `picture` but none of the media definitions match.

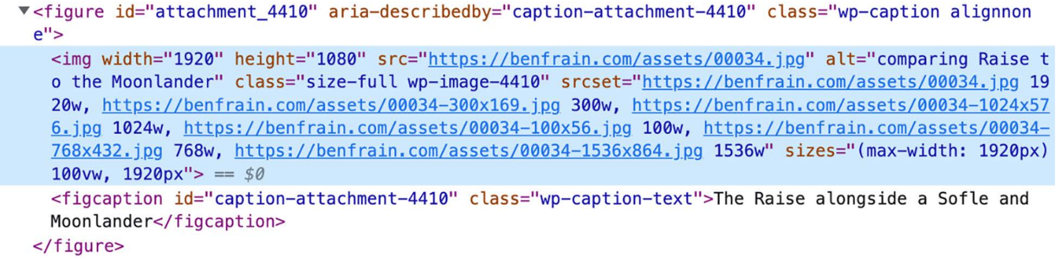
> Just to be crystal clear, do not omit the `img` tag from within a `picture` element or things won't end well!

The key difference with `picture` is that we have a `source` tag. Here, we can use media query-style expressions to explicitly tell the browser which asset to use in a matching situation.

These sources are evaluated top to bottom, so if the first query is true in the first source, that is the image that will be shown; if not it goes to the next. If none are true, the image in the `src` of the `img` tag is shown. You can try this example yourself in `example_10-01`.

A practical note on providing alternate image formats and sizes

Writing a `picture` element every time you want to include alternate versions of an image on a page is an awful lot of boilerplate code. As such, many content management systems and static site generator tools will add responsive image code for you behind the scenes. For example, when you upload an image into WordPress for use in a blog post, it automatically makes alternate sizes and serves those up as options in a `srcset` attribute inside an `img` tag:



```
▼<figure id="attachment_4410" aria-describedby="caption-attachment-4410" class="wp-caption alignnon
 e">
   <img width="1920" height="1080" src="https://benfrain.com/assets/00034.jpg" alt="comparing Raise t
   o the Moonlander" class="size-full wp-image-4410" srcset="https://benfrain.com/assets/00034.jpg 19
   20w, https://benfrain.com/assets/00034-300x169.jpg 300w, https://benfrain.com/assets/00034-1024x57
   6.jpg 1024w, https://benfrain.com/assets/00034-100x56.jpg 100w, https://benfrain.com/assets/00034-
   768x432.jpg 768w, https://benfrain.com/assets/00034-1536x864.jpg 1536w" sizes="(max-width: 1920px)
   100vw, 1920px"> == $0
   <figcaption id="caption-attachment-4410" class="wp-caption-text">The Raise alongside a Sofle and
   Moonlander</figcaption>
 </figure>
```

Figure 10.2: The kind of markup WordPress adds when I insert an image into my blog

My advice would be to avoid "hand-writing" the `picture` element and `srcset` code as much as possible. If you are using a content management system that can do it for you, great. If not, at least defer it to a modern AI assistant, which can spit the relevant code out for you easily, and likely with less errors.

Summary

Hopefully, the terminology, techniques, and appropriate applications for responsive images are now a little clearer.

In this chapter, we looked at newer image formats such as AVIF and WebP, and the massive economies they can offer in terms of file size. We looked at how we can use the `picture` element to serve up both of those image types alongside a suitable fallback.

We also looked at `srcset` and how that allows us to provide the browser with a number of possibilities alongside our preference, enabling the browser to make an informed choice about which to use.

We then looked at `picture` again, considering how to use it to serve entirely different images based on things such as viewport width.

The techniques we have covered in this chapter apply to bitmap images: photos and such. But that's just one side of the image coin. In the next chapter, we will take a deep dive into **Scalable Vector Graphics** (**SVG**), and how to embrace the vector `image` format made for the web.

11

SVG

SVG is an important technology for responsive web design as it provides pin-sharp and future-proof graphical assets for all screen resolutions.

Images on the web with formats such as JPEG, GIF, or PNG are known as raster or Bitmap images. They have their visual data saved as set pixels. If you save a graphic in any of those formats with a set width and height and zoom the image to twice its original size or more, the limitations of the formats can be easily exposed.

Here’s a screenshot of just that, a PNG image I’ve zoomed into in the browser:

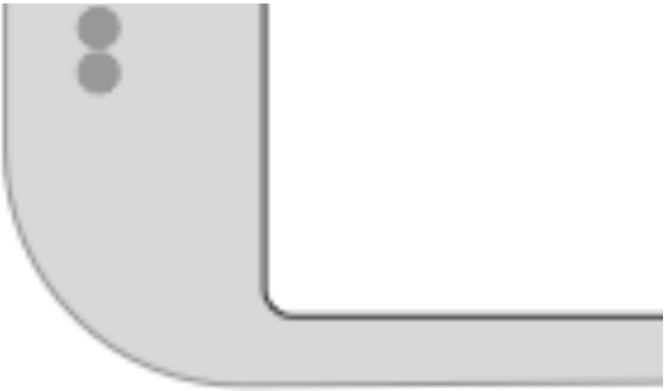


Figure 11.1: The shortcomings of raster images are easy to see on today’s high-definition screens

Can you see how the image looks obviously pixelated? Here is the exact same image saved as a vector image, in SVG format, and zoomed to a similar level:

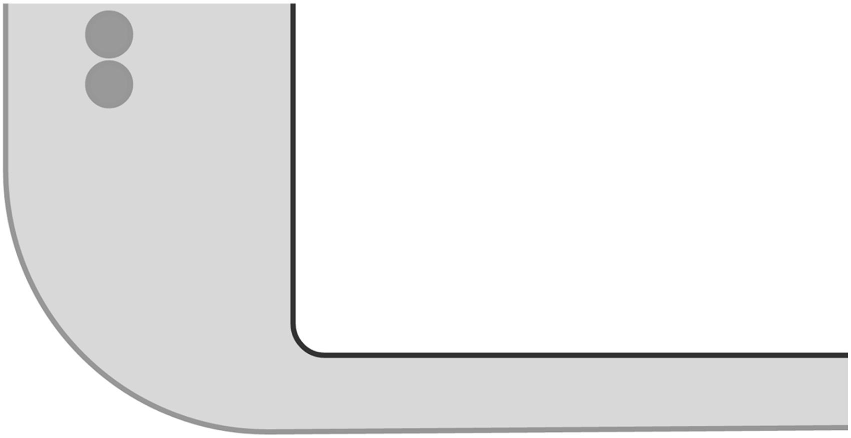


Figure 11.2: An SVG looks sharp, regardless of the display size

Hopefully, the difference is obvious.

Unless you are dealing with photographic imagery, using SVG rather than JPEG, GIF, or PNG for any kind of vector artwork will produce resolution-independent graphics that require far smaller file sizes.

While we will touch upon many aspects of SVG in this chapter, the focus will be on how to integrate them into your workflow, while also providing an overview of what is possible with SVG.

In this chapter, we will cover the following topics:

- SVG: A brief history and the anatomy of a basic SVG document
- Creating SVGs with popular image-editing packages and services
- Inserting SVGs into a page with `img` and `object` tags
- Inserting SVGs as background images
- Inserting SVGs directly (inline) into HTML
- Reusing SVG symbols
- Referencing external SVG symbols
- What capabilities are possible with each insertion method
- Animating SVGs with SMIL
- Styling SVGs with an external style sheet

- Styling SVGs with internal styles
- Re-coloring and animating SVGs with CSS
- Media queries and SVGs
- Optimizing SVGs
- Using SVGs to define filters for CSS
- Manipulating SVGs with JavaScript and JavaScript libraries
- Implementation tips
- Further resources

SVG is a dense subject. Which portions of this chapter are most relevant to your needs will depend on what you actually need from SVG. I can offer a few shortcuts up front.

If you simply want to replace static graphical assets on a website with SVG versions, for sharper images and/or smaller file sizes, then look at the shorter sections on using SVG as background images and within `img` tags.

If you're curious about what applications and services can help you generate and manage SVG assets, skip to the *Creating SVGs with popular image-editing packages and services* section for some useful links and pointers.

If you want to understand SVG more fully, or animate and manipulate SVGs, get comfy and grab a double-size of your favorite beverage, as this may take a while.

To begin our journey of understanding, step back with me into 2001.

A brief history of SVG

The first release of SVG was in 2001. That was not a typo. SVG has been "a thing" since 2001. While it gained traction along the way, it's only since the advent of high-resolution devices that SVGs have received widespread interest and adoption. Here is the introduction to SVGs from the 1.1 specification (`https://www.w3.org/TR/SVG11/intro.html`):

> *SVG is a language for describing two-dimensional graphics in XML [XML10]. SVG allows for three types of graphic objects: vector graphic shapes (for paths consisting of straight lines and curves), images, and text.*

As the name implies, **Scalable Vector Graphics** (**SVG**) allows two-dimensional images to be described in code as vector points. This makes them a great candidate for icons, line drawings, and charts.

As vectors describe relative points, they can scale to any size, without loss of fidelity. Furthermore, in terms of data, as SVGs are described as vector points, it tends to make them tiny, compared to a comparably sized JPEG, GIF, or PNG file.

Browser support for SVG is now widespread: `https://caniuse.com/#search=svg`.

An image that is also a readable web document

Ordinarily, if you view the code of an image file in a text editor, the resultant text is completely unintelligible.

Where SVG graphics differ is that they are actually described in a markup style language. SVG is written in **Extensible Markup Language (XML)**, a close relative of HTML. Although you may not realize it, XML is everywhere on the internet. Do you use an RSS reader, or listen to Podcasts? That's XML right there. XML is the language that wraps up the content of RSS and Podcast feeds, and makes them consumable to other tools and services.

So, not only can machines read and understand SVG graphics, but we can too.

Let me give you an example. Take a look at this star graphic:

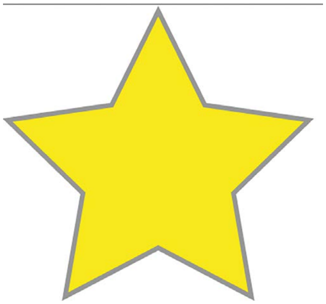


Figure 11.3: A basic SVG

This is an SVG graphic, called `Star.svg`, inside `example_11-01`. You can either open this example in the browser, where it will appear as the star, or you can open it in a text editor, and you can see the code that generates it. Consider this:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg
  width="198px"
  height="188px"
  viewBox="0 0 198 188"
  version="1.1"
```

```
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:sketch="http://www.bohemiancoding.com/sketch/ns"
  >
    <!-- Generator: Sketch 3.2.2 (9983) - http://www.bohemiancoding.com/
    sketch -->
    <title>Star 1</title>
    <desc>Created with Sketch.</desc>
    <defs></defs>
    <g
      id="Page-1"
      stroke="none"
      stroke-width="1"
      fill="none"
      fill-rule="evenodd"
      sketch:type="MSPage"
    >
      <polygon
        id="Star-1"
        stroke="#979797"
        stroke-width="3"
        fill="#F8E81C"
        sketch:type="MSShapeGroup"
        points="99 154 40.2214748 184.901699 51.4471742 119.45085
          3.89434837 73.0983006 69.6107374 63.5491503 99 4
            128.389263 63.5491503 194.105652 73.0983006 146.552826
              119.45085 157.778525 184.901699 "
      ></polygon>
    </g>
  </svg>
```

That is the entirety of the code needed to generate that star as an SVG graphic.

Now, ordinarily, if you've never looked at the code of an SVG graphic before, you may be wondering why you would ever want to. If all you want is vector graphics displayed on the web, you certainly don't need to. Just find a graphics application that will save your vector artwork as an SVG and you're done.

We will list a few of those packages in the coming pages. However, although it's certainly common and possible to only work with SVG graphics from within a graphics-editing application, understanding exactly how an SVG fits together and how you can tweak it to your exact will can become very useful if you need to start manipulating and animating an SVG.

So, let's take a closer look at that SVG markup and get an appreciation of what exactly is going on in there. I'd like to draw your attention to a few key things.

The root SVG element

The root SVG element here has attributes for `width`, `height`, and `viewBox`:

```
<svg width="198px" height="188px" viewBox="0 0 198 188"
```

Each of these plays an important role in how an SVG is displayed.

Hopefully, at this point, you understand the term **viewport**. It's been used in most chapters of this book to describe the area of a device through which content is viewed. For example, a mobile device might have a 320 px-by-480 px viewport. A desktop computer might have a 1,920 px-by-1,080 px viewport.

The `width` and `height` attributes of the SVG effectively create a viewport. Through this defined viewport, we can peek in to see the shapes defined inside the SVG. Just like a web page, the contents of the SVG may be bigger than the viewport, but that doesn't mean the rest isn't there; it's merely hidden from our current view.

`viewBox`, on the other hand, defines the coordinate system in which all the shapes of the SVG are governed. You can think of the `viewBox` values `0 0 198 188` as describing the top-left and bottom-right areas of a rectangle. The first two values, known technically as `min-x` and `min-y`, describe the top-left corner, while the second two, known technically as `width` and `height`, describe the bottom-right corner. Having the `viewBox` attribute allows you to do things such as zoom an image in or out. For example, you can halve the `width` and `height` values in the `viewBox` attribute like this:

```
<svg width="198px" height="188px" viewBox="0 0 99 94"
```

The shape, which is half the size of the SVG, will "zoom" up to fill the size of the SVG width and height.

To really understand `viewBox` and the SVG coordinate system and the opportunities it presents, I recommend this post by Sara Soueidan: `https://sarasoueidan.com/blog/svg-coordinate-systems/` and this post by Jakob Jenkov: `https://jenkov.com/tutorials/svg/index.html`.

Namespace

This SVG has an additional namespace defined for the Sketch graphics program that generated it (`xmlns` is short for **XML namespace**):

```
xmlns:sketch="http://www.bohemiancoding.com/sketch/ns"
```

These namespace references tend to only be used by the program that generated the SVG, so they are often not needed when the SVG is bound for the web. Optimization processes for reducing the size of SVGs will often strip them out.

The title and desc tags

There are `title` and `desc` tags that make an SVG document highly accessible:

```
<title>Star 1</title>
  <desc>Created with Sketch.</desc>
```

These tags can be used to describe the contents of the graphics when they cannot be seen. However, when SVG graphics are used for background graphics, these tags can be stripped to further reduce the file size.

The defs tag

There is an empty `defs` tag in our example code:

```
<defs></defs>
```

Despite being empty in our example, this is an important element. It is used to store definitions of all manner of reusable content, such as gradients, symbols, and paths.

The g element

The g element is used to group other elements together. For example, if you were drawing an SVG of a car, you might group the shapes that make up an entire wheel inside a g tag:

```
<g id="Page-1" stroke="none" stroke-width="1" fill="none"
  fill-rule="evenodd" sketch:type="MSPage">
```

In our g tag, we can see the earlier namespace of sketch being reused here. It will help that graphics application open this graphic again, but it serves no further purpose should this image be bound for elsewhere.

SVG shapes

The innermost node in this example is a polygon:

```
<polygon id="Star-1" stroke="#979797" stroke-width="3" fill="#F8E81C"
  sketch:type="MSShapeGroup" points="99 154 40.2214748 184.901699
    51.4471742 119.45085 3.89434837 73.0983006 69.6107374 63.5491503
      99 4 128.389263 63.5491503 194.105652 73.0983006 146.552826
        119.45085 157.778525 184.901699 ">
</polygon>
```

SVGs have a number of ready-made shapes available: `path`, `rect`, `circle`, `ellipse`, `line`, `polyline`, and `polygon`.

SVG paths

SVG paths differ from the shapes of SVGs as they are composed of any number of connected points, giving you the freedom to create any shape you like.

For example, take a look at this `path`, which describes the same star as before but with the shape described as a path instead of a polygon. Pay particular attention to the `d` attribute:

```
<path d="M99 154l-58.78 30.902 11.227-65.45L3.894 73.097l65.717-9.55L99
  4l29.39 59.55 65.716 9.548-47.553 46.353 11.226 65.452z" />
```

Remember in *Chapter 9*, when we looked at the `shape()` function? The `d` attribute is the part we take and convert to display something with `shape()`.

So, those are the guts of an SVG file, and hopefully you now have a high-level understanding of what's going on. While some will relish the opportunity to handwrite or edit SVG files in code, a great many more would rather generate SVGs with a graphics package. Let's consider some of the more popular choices.

Creating SVGs with popular image-editing packages and services

While SVGs can be opened, edited, and written in a text editor, there are plenty of applications that offer a **Graphical User Interface** (**GUI**) that make authoring complex SVG graphics easier. Perhaps the most obvious choice is Adobe's Illustrator (PC/Mac). However, it is expensive for casual users, so my own preference is either Bohemian Coding's Sketch (Mac only: `https://www.sketch.com/`), which admittedly still isn't cheap at $120 a year, or Figma, which is cross-platform and also uses a subscription-based model that is currently free for the starter plan.

Check out Figma at `https://www.figma.com/`. Or, if one-off payments are your preference, look at Affinity Designer (`https://affinity.serif.com/en-gb/designer/`), which supports Windows and Mac (no Linux, sadly) and costs $69.

If budget is the main concern, consider the free and open source Inkscape (`https://inkscape.org`). It's by no means the prettiest tool to work with, but it is very capable (if you want any proof, view the Inkscape gallery: `https://inkscape.org/gallery/`).

Finally, there are a few online editors. Google has svg-edit (`https://github.com/SVG-Edit/svgedit`). There is also DRAW SVG (`https://www.drawsvg.org/`) and Method Draw, an arguably better-looking fork of svg-edit (`https://editor.method.ac/`).

Saving time with SVG icon services

The aforementioned applications all give you the capability to create SVG graphics from scratch. However, if it's icons you're after, you can probably save a lot of time (and, from my perspective, get better results) by downloading SVG versions of icons from an online icon service.

My personal favorite is `https://icomoon.io`, although `https://fontastic.me` is also great. To quickly illustrate the benefits of an online icon service, loading the `https://icomoon.io` application gives you a searchable library of icons (some free, some paid):

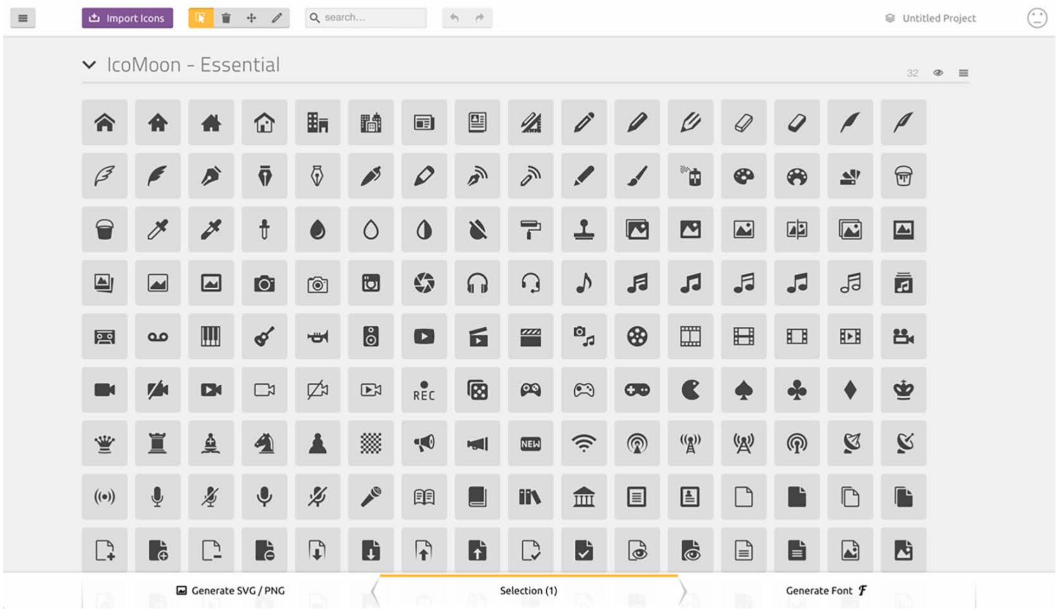


Figure 11.4: Online services can make it simple to get started incorporating SVGs

You select the ones you want and then click **Download**. The resultant file contains the icons as SVGs, PNGs, and also SVG symbols for placement in the `defs` element (remember that the `defs` element is a container element for referenced elements).

To see for yourself, open `example_11-02`. You can see the resultant downloaded files after I chose five icons from `https://icomoon.io`.

Inserting SVGs into your web pages

There are a number of things that you can do with SVG images that you can't do with normal image formats (JPEG, GIF, and PNG). The range of what's possible is largely dependent on the way that the SVG is inserted into the page.

So, before we get to what we can actually do with SVGs, we'll consider the various ways we can actually get them on the page in the first place.

Using an img tag

The most straightforward way to use an SVG graphic is exactly how you would insert any image into an HTML document. We just use a good ol' `img` tag:

```
<img src="mySconeVector.svg" alt="Amazing line art of a scone" />
```

This makes the SVG behave more or less like any other image. There isn't much else to say about that.

Using an object tag

The `object` tag is the container recommended by the W3C for holding non-HTML content in a web page (the specification for `object` is here: `https://html.spec.whatwg.org/multipage/iframe-embed-object.html#the-object-element`). We can make use of it to insert an SVG into our page like this:

```
<object data="img/svgfile.svg" type="image/svg+xml">
  <span class="fallback-info">Your browser doesn't support SVG</span>
</object>
```

Either a `data` or `type` attribute is required, although I would always recommend adding both. The `data` attribute is where you link out to the SVG asset in the same manner you would link to any other asset. The `type` attribute describes the MIME (internet media type) type relevant to the content. In this instance, `image/svg+xml` is the MIME type to indicate that the data is SVG. You can add a `width` and `height` attribute too if you want to constrain the size of the SVG with this container.

An SVG inserted into the page via an `object` tag is also accessible with JavaScript, so that's another point worth considering. There is a built-in fallback mechanism with `object`, too; if the browser does not understand the type of the object, it can render the span inside. So, while there are no browsers in common use today that would struggle with SVG, you might use an `object` tag to hold some other, newer content type and provide some fallback for browsers that don't understand it.

Inserting an SVG as a background image

SVGs can be used as background images in CSS, much the same as any other image format (PNG, JPG, or GIF). There's nothing special about the way you reference them:

```
.item {
  background-image: url('image.svg');
}
```

If your needs for SVG are primarily static background images, perhaps for icons and the like, I recommend implementing SVGs as background images. That's because there are a number of tools that will automatically embed the SVGs as data URIs, or create image sprites as needed. Using SVGs this way is very well supported; the images themselves cache well (so performance-wise, they work very well), and it's simple to implement.

A brief aside on data URIs

After reading the previous section, you might wonder what on earth a data **Uniform Resource Identifier** (**URI**) is. In relation to CSS, it's a means of embedding what would ordinarily be an external asset, such as an image, within the CSS file itself. Therefore, we might ordinarily do this to link to an external image file:

```
.external {
  background-image: url('Star.svg');
}
```

Instead, however, we could include the image inside our style sheet with a data URI, like this:

```
.data-uri {
  background-image: url(data:image/svg+xml,%3C%3Fxml%20
  version%3D%221.0%22%20encoding%3D%22UTF-8%22%20
  standalone%3D%22no%22%3F%3E%0A%3Csvg%20width%3D%22198px%22%20
  height%3D%22188px%22%20viewBox%3D%220%200%20198%20
  188%22%20version%3D%221.1%22%20xmlns%3D%22http%3A%2F%2Fwww.
  w3.org%2F2000%2Fsvg%22%20xmlns%3Axlink%3D%22http%3A%2F%2Fwww.
  w3.org%2F1999%2Fxlink%22%20xmlns%3Asketch%3D%22http%3A%2F%2Fwww.
  bohemiancoding.com%2Fsketch%2Fns%22%3E%0A%20%20%20%20%3C%21--%20
  Generator%3A%20Sketch%203.2.2%20%289983%29%20-%20http%3A%2F%2Fwww.
  bohemiancoding.com%2Fsketch%20--%3E%0A%20%20%20%20%3Ctitle%3EStar%20
  1%3C%2Ftitle%3E%0A%20%20%20%20%3Cdesc%3ECreated%20with%20
  Sketch.%3C%2Fdesc%3E%0A%20%20%20%20%3Cdefs%3E%3C%2Fdefs%3E%0A%20
  %20%20%20%3Cg%20id%3D%22Page-1%22%20stroke%3D%22none%22%20stroke-
  width%3D%221%22%20fill%3D%22none%22%20fill-rule%3D%22evenodd%22%20
  sketch%3Atype%3D%22MSPage%22%3E%0A%20%20%20%20%20%20%20%20%3Cpolygon%20
  id%3D%22Star-1%22%20stroke%3D%22%23979797%22%20stroke-width%3D%223%22%20
  fill%3D%22%23F8E81C%22%20sketch%3Atype%3D%22MSShapeGroup%22%20
  points%3D%2299%20154%2040.2214748%20184.901699%2051.4471742%20
  119.45085%203.89434837%2073.0983006%2069.6107374%2063.5491503%2099%20
  4%20128.389263%2063.5491503%20194.105652%2073.0983006%20146.552826%20
  119.45085%20157.778525%20184.901699%20%22%3E%3C%2Fpolygon%3E%0A%20%20%20
  %20%3C%2Fg%3E%0A%3C%2Fsvg%3E);
}
```

It's not pretty, but it provides us with a way to negate a separate request over the network. There are different encoding methods for data URIs and plenty of tools available to create data URIs from your assets.

> Modern build tools such as Vite (`https://vite.dev`) automatically embed linked SVG assets under a set size with zero configuration.

If you're encoding SVGs in this manner, I would suggest avoiding the Base64 method as it doesn't compress as well as text for SVG content.

The argument against the excessive use of SVG content embedded as a data URI is that your style sheet will be much larger, and that will mean it takes longer to fetch over the network and parse.

Generating image sprites

My personal recommendation, tool-wise, for generating image sprites or data URI assets is iconizr (`https://iconizr.com`). It gives you complete control over how you would like your resultant SVG and fallback PNG assets. You can have the SVGs and fallback PNG files output as data URIs or image sprites, and it even includes the requisite JavaScript snippet for loading the correct asset if you opt for data URIs. This is highly recommended.

While I’m a big fan of SVGs as background images, if you want to animate them dynamically, or inject values into them via JavaScript, then it will be best to opt for inserting SVG data “inline” into the HTML.

Inserting an SVG inline

As SVG is merely an XML document, you can insert it directly into the HTML. Take the following example:

```
<div>
  <h3>Inserted 'inline':</h3>
  <span class="inlineSVG">
    <svg id="svgInline" width="198" height="188" viewBox="0 0 198 188"
      xmlns="http://www.w3.org/2000/svg"
        xmlns:xlink="http://www.w3.org/1999/xlink">
    <title>Star 1</title>
      <g class="star_Wrapper" fill="none" fill-rule="evenodd">
        <path id="star_Path" stroke="#979797" stroke-width="3"
          fill="#F8E81C" d="M99 154l-58.78 30.902 11.227-
            65.45L3.894 73.097l65.717-9.55L99 4l29.39 59.55
              65.716 9.548-47.553 46.353 11.226 65.452z" />
      </g>
    </svg>
  </span>
</div>
```

There is no special wrapping element needed; you literally just insert the SVG markup inside the HTML markup. It’s also worth noting that if you remove any `width` and `height` attributes on the `svg` element, the SVG will scale fluidly to fit the containing element.

Inserting SVG as markup into your documents is probably the most versatile option in terms of accessing the broadest range of SVG features.

Let’s consider some of those features that inserting SVGs inline allows next.

Reusing graphical objects from symbols

Earlier in this chapter, I mentioned that I had picked and downloaded some icons from IcoMoon (`https://icomoon.io`). They were icons depicting touch gestures: swipe, pinch, drag, and more. Suppose, in a website you are building, you need to make use of them multiple times. Remember how I mentioned that there was a version of those icons as SVG symbol definitions? That’s what we will make use of now.

In `example_11-03`, we insert the various symbol definitions inside the `defs` element of an SVG in the page. You'll notice that, on the `svg` element, an inline style is used, `display:none`, and the `height` and `width` attributes have both been set to `0` (those styles could be set in CSS if you prefer). This is so that this SVG takes up no space. We are only using this SVG to house symbols of the graphical objects we want to use elsewhere. So, our markup starts like this:

```
<body>
  <svg display="none" width="0" height="0" version="1.1"
    xmlns="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xlink">
  <defs>
  <symbol id="icon-drag-left-right" viewBox="0 0 1344 1024">
    <title>drag-left-right</title>
    <path class="path1" d="M256 192v-160l-224 224 224
      224v-160h256v-128z"></path>
```

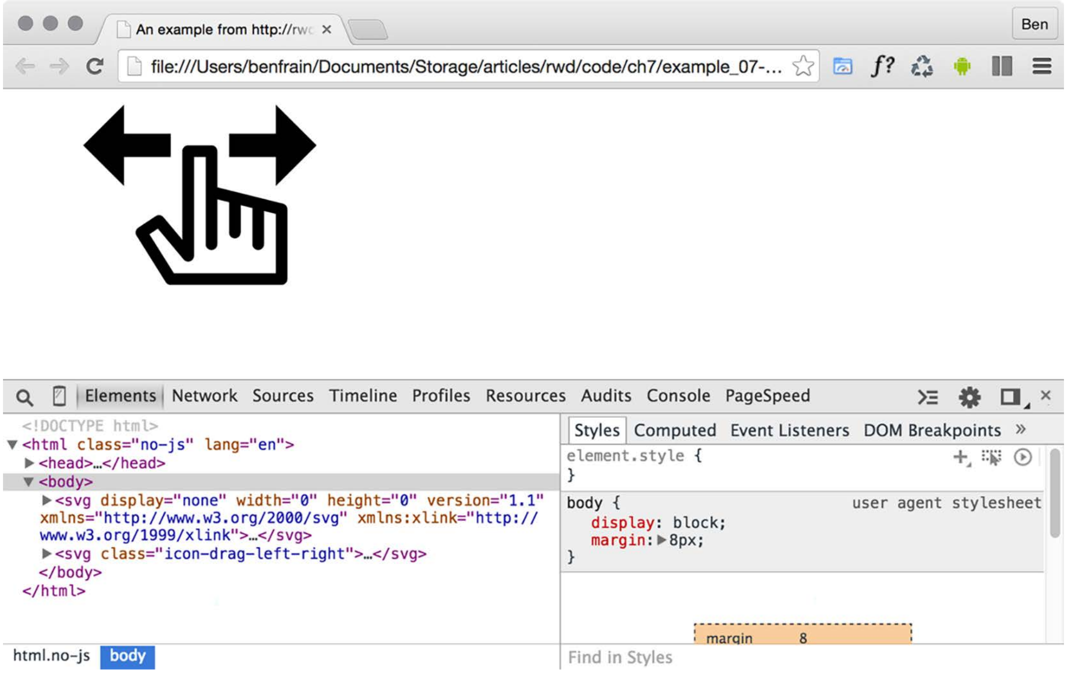
Notice the `symbol` element inside the `defs` element? This is the element to use when we want to define a shape for later reuse.

You can create symbols for use inside the `defs` tag by taking an existing SVG, changing the `svg` tag to `symbol`, removing the `namespace` attribute, and then nesting it inside the `defs` tag.

After the SVG has defined all necessary symbols for our work, we have all our "normal" HTML markup. Then, when we want to make use of one of those symbols, we can do this:

```
<svg class="icon-drag-left-right">
  <use xlink:href="#icon-drag-left-right"></use>
</svg>
```

That will display the drag left and right icon:


An example from http://rwc ×
Ben
file:///Users/benfrain/Documents/Storage/articles/rwd/code/ch7/example_07-...
Elements
Network
Sources
Timeline
Profiles
Resources
Audits
Console
PageSpeed
Styles
Computed
Event Listeners
DOM Breakpoints
margin
8
html.no-js
body
Find in Styles

```
<!DOCTYPE html>
▼<html class="no-js" lang="en">
  ▶<head>…</head>
  ▼<body>
    ▶<svg display="none" width="0" height="0" version="1.1"
    xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://
    www.w3.org/1999/xlink">…</svg>
    ▶<svg class="icon-drag-left-right">…</svg>
    </body>
  </html>
```

```
element.style {
}
body {                                    user agent stylesheet
    display: block;
    margin:▶8px;
}
```

Figure 11.5: With SVGs inserted inline, it's possible to reuse graphics

The magic here is the use element. As you might have guessed from the name, it's used to make use of existing graphical objects that have already been defined elsewhere. The mechanism for choosing what to reference is the `xlink` attribute, which, in this case, is referencing the symbol ID of the "drag left and right" icon (`#icon-drag-left-right`) we have inline at the beginning of the markup.

When you reuse a symbol, unless you explicitly set a size (either with attributes on the element itself or with CSS), use will be set to a width and height of 100%. So, to resize our icon, we could do this:

```
.icon-drag-left-right {
  width: 2.5rem;
  height: 2.5rem;
}
```

The use element can be used to reuse all sorts of SVG content: gradients, shapes, symbols, and more.

Inline SVGs allow different colors in different contexts

With inline SVGs, you can also do useful things such as change colors based on context, and that's great when you need multiple versions of the same icon in different colors:

```
.icon-drag-left-right {
  fill: #f90;
}
.different-context .icon-drag-left-right {
  fill: #ddd;
}
```

Making dual-tone icons that inherit the color of their parent

With inline SVGs, you can also have some fun and create two-tone effects from a single color icon (as long as the SVG is made up of more than one path) with the use of `currentColor`, the oldest CSS variable. To do this, inside the SVG symbol, set the `fill` attribute of the path that you want to be one color as `currentColor`. Then, use the `color` value in your CSS to color the element. For the paths in the SVG symbol without the fill, set them as `currentColor`; they will receive the `fill` value. Here is an example:

```
.icon-drag-left-right {
  width: 2.5rem;
  height: 2.5rem;
  fill: #f90;
  color: #ccc; /* this gets applied to the path that has its fill
  attribute set to currentColor in the symbol */
}
```

Here's that same symbol reused three times, each with different colors and sizes:

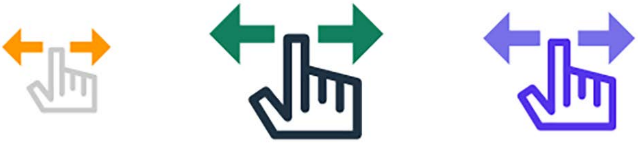


Figure 11.6: Making dual-color SVGs by inheriting colors

Remember, you can dig around the code in `example_11-03`. It's also worth noting that `color` doesn't have to be set on that element itself—it can be on any parent element; `currentColor` will inherit a value from up the DOM tree to the nearest parent with a `color` value set.

Recoloring SVGs with CSS custom properties

Nowadays, I find the most effective way to recolor the same SVG or symbol is by using CSS custom properties. We explore them fully in *Chapter 13*, *Custom Properties and CSS Functions*, but for now, we will just consider how they can help us with SVG. The following example can be found at `example_11-04`.

I have made a pentagon shape using two paths. Each path has a `fill` and `stroke` value. Here is the SVG:

```
<svg class="shape" width="268" height="254" xmlns="http://www.w3.org/2000/
svg">
  <g fill="none" fill-rule="evenodd" stroke-width="10">
    <path d="M134 6.18L6.73 98.647l48.613 149.615h157.314L261.27
      98.647 134 6.18z" stroke="var(--stroke1)"
        fill="var(--fill1)"/>
    <path d="M134 36.18l-98.738 71.738 37.714 116.074h122.048l37.714-
      116.074L134 36.18z" stroke="var(--stroke2)"
        fill="var(--fill2)"/>
  </g>
</svg>
```

Notice how for each `path` element, both the `stroke` and `fill` values have been set to a CSS custom property. With those values in place, I can then set any colors I want for that SVG with CSS, JavaScript, or a combination of the two!

In our example, I have set default colors and hover colors, like this:

```
.shape {
  display: block;
  --stroke1: #ddd;
  --fill1: #444;
  --stroke2: #f90;
  --fill2: #663d00;
}
.shape:hover {
  --stroke1: #333;
  --fill1: #444;
  --stroke2: #fff;
  --fill2: #ffc266;
}
```

This is the default coloring (I made the example document background dark just to make it clearer):

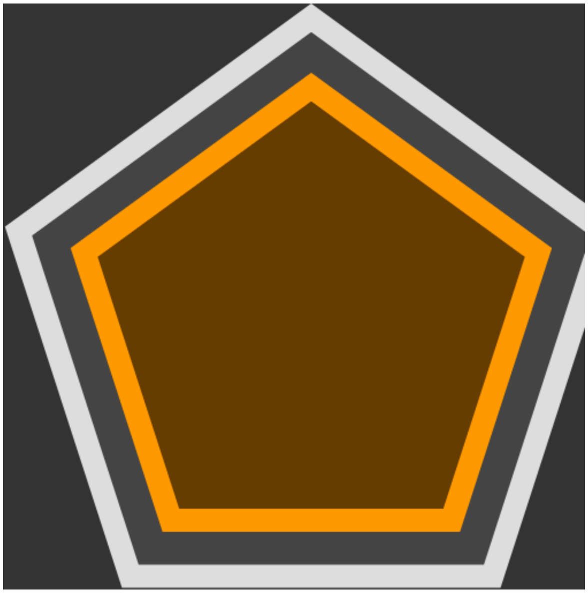


Figure 11.7: CSS custom properties make recoloring SVGs simple

Then, to prove how easy it is to amend it further still, if you open the example, you will see a button that uses a tiny bit of JavaScript to toggle a class on the page, adding or removing the `.amended` class. This allows us to have two further sets of colors for this shape:

```
.amended .shape {
  --stroke1: #080b2b;
  --fill1: #141a67;
  --fill2: #192183;
  --stroke2: #2d3ad7;
}
.amended .shape:hover {
  --stroke1: #092b08;
  --fill1: #125610;
  --fill2: #2dd728;
  --stroke2: #1b8118;
}
```

This results in four blue colors ordinarily and four green colors when hovered over:

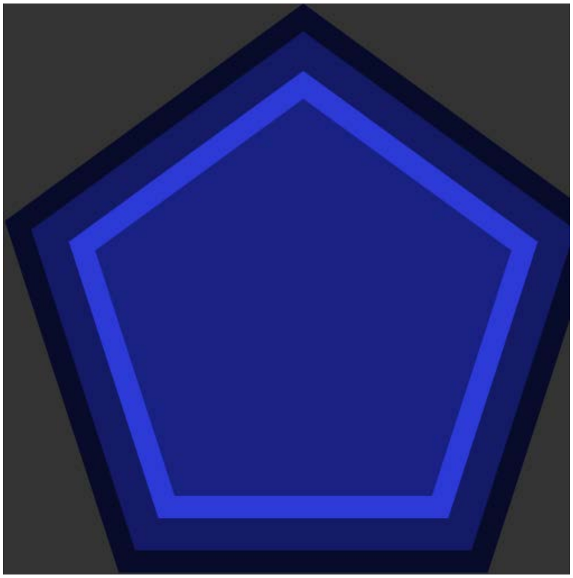


Figure 11.8: CSS custom properties create near-limitless possibilities

As you can see, there are a lot of positives to placing SVGs inside the markup. The only downside is that it's necessary to include the same SVG data on every page where you want to use the icons. Sadly, this is bad for performance, as the assets (the SVG data) aren't going to be cached easily. However, there is another option.

Reusing graphical objects from external sources

Rather than paste an enormous set of SVG symbols on each page, while still using the `use` element, it's possible to link out to external SVG files and grab the portion of the document you want to use. Take a look at `example_11-05` and the same three icons as we had in `example_11-03`, which are put on the page in this manner:

```
<svg class="icon-drag-left-right">
  <use xlink:href="defs.svg#icon-drag-left-right"></use>
</svg>
```

Be aware that some of these examples need to be run on a server. There are two simple solutions: either use something such as Browsersync or `npx serve` to create a simple local server, or paste the code into a service such as `https://codepen.io/`.

The important part to understand is `href`. We are linking to an external SVG file (the `defs.svg` part) and then specifying the ID of the symbol within that file that we want to use (the `#icon-drag-left-right` part). The benefits of this approach are that the asset is cached by the browser (just like any other external image would/could be) and it saves littering our markup with an SVG full of symbol definitions. The downside is that, unlike when `defs` is placed inline, any dynamic changes to `defs.svg` (for example, if a path were being manipulated by JavaScript) won't be updated in the use tags.

Recoloring SVG assets with a mask

It's common, in large projects, to want to use the same symbol but in a different color in a different context. An SVG inserted into the page is the best option here. It affords the ability for any part or whole of the SVG to be recolored. However, that may not be practical or preferable for any number of reasons. You should know that if your SVGs are a single color, you can recolor them using a mask. Consider our star SVG inserted into the page as a background image:

```
<div class="mask-bg"></div>
```

However, instead of using `background-image` to show the star image, we can use `mask-image`. The markup stays as before but the CSS looks like this:

```
.mask-bg {
  /* background-image: url("Star.svg"); */
  /* background-repeat: no-repeat; */
  mask-image: url("Star.svg");
  mask-repeat: no-repeat;
  background-color: green;
  height: 200px;
  width: 200px;
  transition: background-color 1s;
  &:hover {
    background-color: hotpink;
  }
}
```

If you take a look at this in the browser, as with `example_11-06`, you can see that not only do you get the star in the color specified, but you can also get a different color on hover with a transition.

Using a mask on single-color SVGs to recolor them is very effective and simple to reason about in a code base. The `mask-*` properties all mirror the equivalent `background-*` properties, so they are easy to remember.

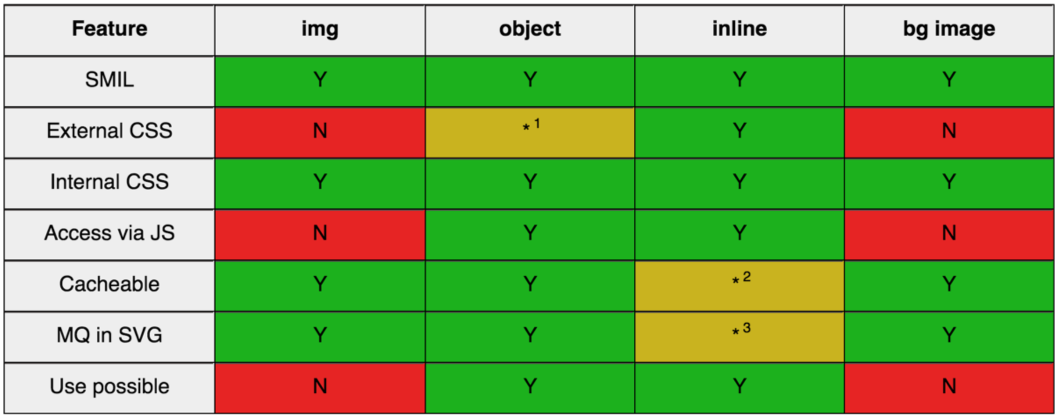
We covered masks a little more in *Chapter 9*, *Stunning Aesthetics with CSS*.

What you can do with each SVG insertion method (inline, object, background-image, and img)

As mentioned previously, SVGs differ from other graphical assets. They can behave differently depending on the way they are inserted into a page. As we have seen, there are four main ways in which to place an SVG onto a page:

- Inside an `img` tag
- Inside an `object` tag
- As a background image
- Inline

Depending on the insertion method, certain capabilities will or will not be available to you. As we go on to look at some of these capabilities, it may be useful to refer back to this table, which covers what's possible with each insertion method:



Feature	img	object	inline	bg image
SMIL	Y	Y	Y	Y
External CSS	N	*[1]	Y	N
Internal CSS	Y	Y	Y	Y
Access via JS	N	Y	Y	N
Cacheable	Y	Y	*[2]	Y
MQ in SVG	Y	Y	*[3]	Y
Use possible	N	Y	Y	N

Figure 11.9: Depending on the insertion method, differing capabilities are possible

Now, there are caveats to consider, marked with numbers in the preceding table:

1. When using an SVG inside an object, you can use an external style sheet to style the SVG, but you have to link to that style sheet from within the SVG.
2. You can use SVGs in an external asset (which is cacheable), although those external assets cannot be manipulated by JavaScript.
3. A media query inside the `styles` section of an "inlined" SVG works on the size of the document it lives in (not the size of the SVG itself).

Be aware that browser implementations of SVG also vary. Therefore, just because those things should be possible (as indicated previously), doesn't mean they will be in every browser, or that they will behave consistently!

The results in the preceding table are based on the test page in `example_11-07`.

The behavior of the test page was comparable in the latest versions of Chrome, Firefox and Safari when tested in summer 2025.

Extra SVG capabilities and oddities

Let's put aside the foibles of browsers for a moment and consider what some of these features in the table actually allow, and why you may or may not want to make use of them.

SVGs will always render as sharp as the viewing device will allow and regardless of the manner of insertion. For most practical situations, resolution independence is usually reason enough to use SVG. It's then just a question of which insertion method would best suit your workflow and the task at hand.

However, there are other capabilities and oddities that are worth knowing about, such as SMIL animation, different ways to link to external style sheets, marking internal styles with character data delimiters, amending an SVG with JavaScript, and making use of media queries within an SVG. Let's cover those next.

SMIL animation

SMIL animations (`https://www.w3.org/TR/smil-animation/`) are a way to define animations for an SVG within the SVG document itself. **SMIL** (pronounced *smile*, in case you were wondering) stands for **Synchronized Multimedia Integration Language** and was developed as a method of defining animations inside an XML document (remember, SVG is XML-based).

Here's an example of how to define a SMIL-based animation:

```
<g class="star_Wrapper" fill="none" fill-rule="evenodd">
  <animate
    xlink:href="#star_Path"
    attributeName="fill"
    attributeType="XML"
    begin="0s"
    dur="2s"
    fill="freeze"
    from="#F8E81C"
    to="#14805e"
    />
    <path
      id="star_Path"
      stroke="#979797"
      stroke-width="3"
      fill="#F8E81C"
      d="M99 154l-58.78 30.902 11.227-65.45L3.894 73.097l65.717-9.55L99
        4l29.39 59.55 65.716 9.548-47.553 46.353 11.226 65.452z"
    />
  </g>
```

I've grabbed a section of the earlier SVG we looked at. `g` is a grouping element in SVG, and this one includes both a star shape (the `path` element with `id="star_Path"`) and the SMIL animation within the `animate` element. That simple animation tweens the fill color of the star from yellow to green in two seconds.

Tweening

In case you didn't already know (I didn't!), "tweening" is simply a shortened version of "inbetweening," as it merely indicates all the in-between stages from one animation point to another.

What's more, it does that whether the SVG is put on the page using `img`, `object`, `background-image`, or inline (honestly—open up `example_11-07` in any recent browser to see).

Wow! Great, right? Well, kind of. As a technology, it's been largely ignored for some time, in favor of more direct manipulation methods such as CSS or using JavaScript.

However, if you still need to use SMIL, Sara Soueidan wrote an excellent, in-depth article about SMIL animations here: `https://css-tricks.com/guide-svg-animations-smil/`.

Styling an SVG with an external style sheet

It's possible to style an SVG with CSS. This can be CSS enclosed in the SVG itself, or in the CSS style sheets you would write all your "normal" CSS in.

Now, if you refer back to our features table from earlier in this chapter, you can see that styling SVG with external CSS isn't possible when the SVG is included via an `img` tag or as a background image (apart from in the now-deprecated Internet Explorer). It's only possible when SVGs are inserted via an `object` tag or inline.

There are two syntaxes for linking to an external style sheet from an SVG. The most straightforward way is like this (you would typically add this in the `defs` section):

```
<link href="styles.css" type="text/css" rel="stylesheet"/>
```

It's akin to the way we used to link to style sheets prior to HTML5 (for example, note the `type` attribute is no longer necessary in HTML5). However, despite this working in many browsers, it isn't the way the specifications define how external style sheets should be linked in SVG (`https://www.w3.org/TR/SVG/styling.html#ReferencingExternalStyleSheets`). Here is the correct/official way, actually defined for XML back in 1999 (`https://www.w3.org/1999/06/REC-xml-stylesheet-19990629/`):

```
<?xml-stylesheet href="styles.css" type="text/css"?>
```

You need to add this above the opening SVG element in your file. Take the following example:

```
<?xml-stylesheet href="styles.css" type="text/css"?>
<svg
  width="198"
  height="188"
  viewBox="0 0 198 188"
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink"
></svg>
```

You don't have to use an external style sheet; you can use inline styles directly in the SVG itself if you would rather.

Styling an SVG with internal styles

You can place styles for an SVG within the SVG itself. They should be placed within the `defs` element. As SVG is XML-based, it's safest to include the **Character Data (CDATA)** marker. CDATA simply tells the browser that the information within the CDATA delimited section could ordinarily be interpreted as XML markup, but should not be. The syntax is like this:

```
<defs>
  <style type="text/css">
    <![CDATA[
      #star_Path {
        stroke: red;
      }
      ]]>
    </style>
  </defs>
```

SVG properties and values within CSS

Notice the `stroke` property in the prior code block? This isn't a CSS property; it's an SVG property. There are quite a few specific SVG properties you can use in styles (regardless of whether they are declared inline or via an external style sheet). For example, with an SVG, as you have seen in an earlier example, you don't specify `background-color`; instead, you specify `fill`. You don't specify `border`; you specify `stroke-width`.

For the full list of SVG-specific properties, take a look at the specification here: `https://www.w3.org/TR/SVG/styling.html`.

With either inline or external CSS, it's possible to do all the "normal" CSS things you would expect: change an element's appearance, animate and transform elements, and so on.

Animating an SVG with CSS

Let's consider a quick example of adding a CSS animation inside an SVG (remember, these styles could just as easily be in an external style sheet too). Let's take the star example we have looked at throughout this chapter and make it spin. You can look at the finished example in `example_11-08`:

```
<div class="wrapper">
  <svg
    width="198"
    height="188"
    viewBox="0 0 220 200"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink"
  >
    <title>Star 1</title>
    <defs>
      <style type="text/css">
        <![CDATA[
        @keyframes spin {
          0% {
            transform: rotate(0deg);
           }
          100% {
              transform: rotate(360deg);
          }
        }
        .star_Wrapper {
          animation: spin 2s 1s;
          transform-origin: 50% 50%;
        }
        .wrapper {
          padding: 2rem;
          margin: 2rem;
        }
        ]]>
      </style>
      <g id="shape">
        <path fill="#14805e" d="M50 50h50v50H50z" />
        <circle fill="#ebebeb" cx="50" cy="50" r="50" />
      </g>
    </defs>
    <g class="star_Wrapper" fill="none" fill-rule="evenodd">
      <path
        id="star_Path"
        stroke="#333"
        stroke-width="3"
        fill="#F8E81C"
        d="M99 154l-58.78 30.902 11.227-65.45L3.894 73.097l65.717-9.55L99
          4l29.39 59.55 65.716 9.548-47.553 46.353 11.226 65.453z"
      />
    </g>
  </svg>
</div>
```

If you load that example in the browser, after a one-second delay, the star will spin a full circle over the course of two seconds.

Notice how a `transform-origin` value of `50% 50%` has been set on the SVG? This is because, unlike CSS, the default `transform-origin` of an SVG is not `50% 50%` (center in both axes); it's actually `0 0` (top left). Without that property set, the star would rotate around the top-left point.

You can get quite far animating SVGs with CSS animations. However, when you want to add interactivity or synchronize several events, it's generally best to lean on JavaScript. The good news is that there are great libraries that make animating SVGs really easy. Let's look at an example of that now.

Animating SVG with JavaScript

With an SVG inserted into the page via an `object` tag or inline, it's possible to manipulate the SVG directly or indirectly with JavaScript. By indirectly, I mean it's possible with JavaScript to change a class on or above the SVG that would cause a CSS animation to start. Take the following example:

```
svg {
  /* no animation */
}
.added-with-js svg {
  /* animation */
}
```

However, it's also possible to animate an SVG via JavaScript directly.

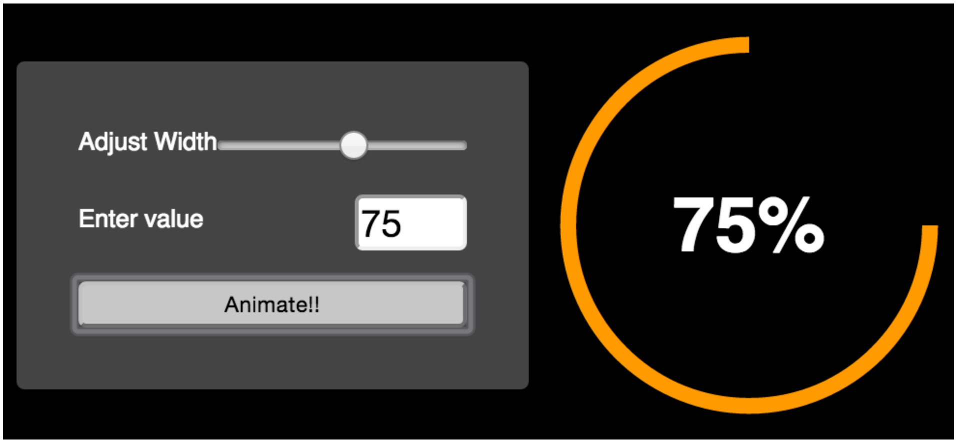
If you're animating just one or two things independently, it's probable that things would be lighter, code-wise, if you wrote the JavaScript by hand. However, if you need to animate lots of elements or synchronize the animation of elements as if on a timeline, JavaScript libraries can really help. Ultimately, you will need to judge whether the weight of including the library in your page can be justified for the goal you are trying to achieve.

My recommendation for animating SVGs via JavaScript is the GreenSock Animation Platform (`https://gsap.com`). For the next example, we'll cover a very simple example using GreenSock.

A simple example of animating an SVG with GreenSock

Suppose we want to make an interface dial that animates when we click a button from zero to whatever value we input. We want not only the stroke of the dial to animate in both length and color but also the number from zero to the value we input. You can view the completed implementation in `example_11-09`.

So, if we entered a value of 75 and clicked **Animate!!**, the dial would fill around to look like this:



Enter value 75

Figure 11.10: Animation libraries make animating SVGs simpler

Instead of listing out the entire JavaScript file (which is heavily commented, so it should make some sense if read in isolation), for brevity's sake, we'll just consider the key points.

The basic idea is that we have made a circle as an SVG `<path>` (rather than a `<circle>` element). As it's a path, it has a stroke. SVG has a property called `stroke-dashoffset`. This value lets you control where the dash applied to a path starts. Because of this, `stroke-dasharray`, which is the actual pattern of the dash, can be offset an amount equal to the length of the path itself. This allows us to animate those properties, and the result is as if the path is being drawn.

There's more info on this technique in the following information box.

> If you haven't come across the SVG "line-drawing" technique before, it was popularized by Polygon magazine when Vox Media animated a couple of line drawings of the Xbox One and PlayStation 4 game consoles. You can read the original post here: `https://product.voxmedia.com/2013/11/25/5426880/polygon-feature-design-svg-animations-for-fun-and-profit`.
>
> There's also an excellent and more thorough explanation of the technique by Jake Archibald here: `https://jakearchibald.com/2013/animated-line-drawing-svg/`.

If the value to animate `dasharray` were a static, known value, this effect would be relatively simple to achieve with a CSS animation and a little trial and error (more on CSS animations in the next chapter). However, besides a dynamic value, at the same time as we are "drawing" the line, we want to fade in the stroke color from one value to another and visually count up to the input value in the text node.

This is an animation equivalent to patting our heads, rubbing our tummies, and counting backward from 10,000. GreenSock makes those things easy (the animation part; it won't rub your tummy or pat your head, although it can count back from 10,000, should you need it to).

Here are the lines of JavaScript needed to make GreenSock do all three actions:

```
// Animate the drawing of the line and color change
TweenLite.to(circlePath, 1.5, {
  'stroke-dashoffset': '-' + amount,
  stroke: strokeEndColour,
});
// Set a counter to zero and animate to the input value
var counter = {var: 0};
TweenLite.to(counter, 1.5, {
  var: inputValue,
  onUpdate: function() {
    text.textContent = Math.ceil(counter.var) + '%';
  },
  ease: Circ.easeOut,
});
```

In essence, with the `TweenLite.to()` function, you pass in the thing you want to animate, the time over which the animation should occur, and then the values you want to change (and what you want them to change to).

The GreenSock site has excellent documentation and support forums, so if you find yourself needing to synchronize a number of animations at once, be sure to clear a day in your diary to familiarize yourself with GreenSock.

Using SVGs as filters

In *Chapter 9*, *Stunning Aesthetics with CSS*, we looked at the CSS filter effects. With help from SVG, we can create even more filters. In `example_11-10`, we have a page with the following markup inside the body:

```
<img class="HRH" src="queen@2x-1024x747.png" />
```

It's an image of the late Queen of England. Ordinarily, it looks like this:

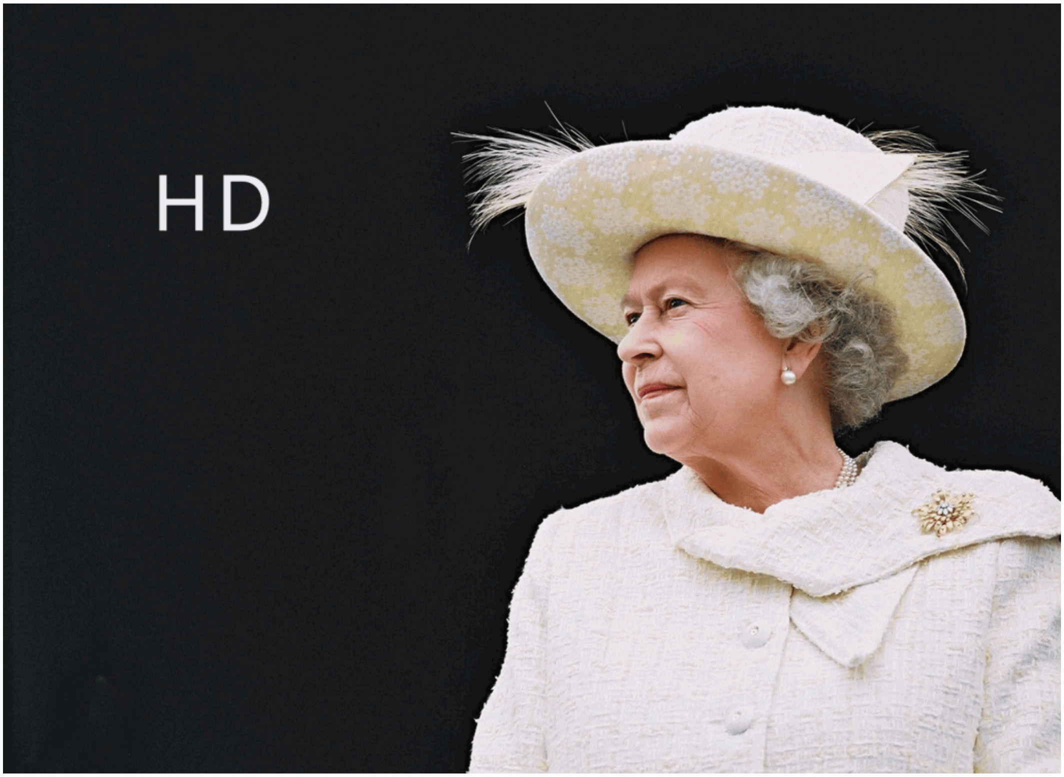


Figure 11.11: An image with no SVG filter applied

Now, also in that example folder is an SVG with a filter defined in the `defs` elements. The SVG markup looks like this:

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <defs>
    <filter id="myfilter" x="0" y="0">
      <feColorMatrix
        in="SourceGraphic"
        type="hueRotate"
        values="90"
        result="A"
      />
      <feGaussianBlur in="A" stdDeviation="6" />
    </filter>
  </defs>
</svg>
```

Within the filter, we are first defining a hue rotation of 90 (using `feColorMatrix`), and then passing that effect, via the `result` attribute, to the next filter (`feGaussianBlur`) with a blur value of 6. Be aware that I've been deliberately heavy-handed here. This doesn't produce a nice aesthetic, but it should leave you in no doubt that the effect has worked!

Now, rather than add that SVG markup to the HTML, we can leave it where it is and reference it using the same CSS filter syntax we saw in the previous chapter:

```
.HRH {
  filter: url('filter.svg#myfilter');
}
```

This is the effect in the browser:

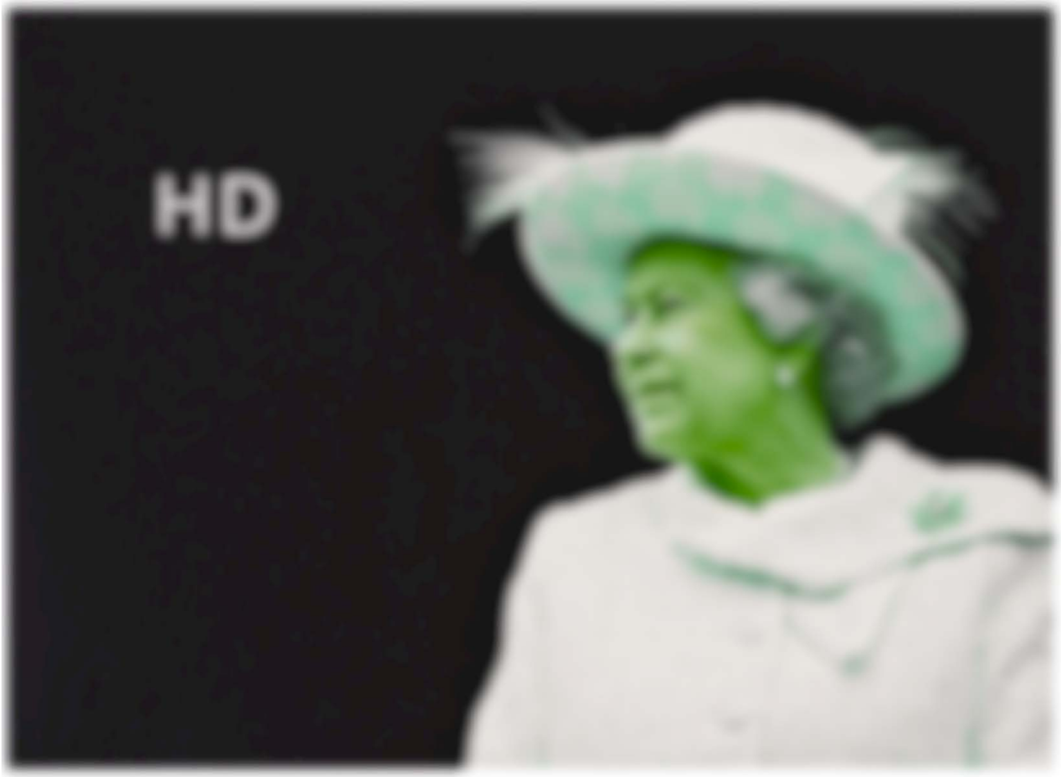


Figure 11.12: An image with an SVG filter applied

So, it's possible with SVG to create for yourself an entirely bespoke filter. Just one more string to SVG's bow!

A note on media queries inside SVGs

All browsers that understand SVG should respect the CSS media queries defined inside. However, when it comes to media queries inside SVGs, there are a few things to remember.

For example, suppose you insert a media query inside an SVG, like this:

```
<style type="text/css"><![CDATA[
  #star_Path {
    stroke: red;
  }
  @media (min-width: 800px) {
    #star_Path {
      stroke: violet;
    }
  }
]]></style>
```

That SVG is displayed on the page at a width of 200 px, inside a browser viewport that is 1,200 px wide.

We might expect the stroke of the star to be violet when the screen is 800 px or above. After all, that's what we have our media query set to. However, when the SVG is placed on the page via an `img` tag, as a background image, or inside an `object` tag, it has no knowledge of the outer HTML document. Hence, in this situation, `min-width` means the minimum width of the SVG itself. So, unless the SVG itself is displayed on the page at a width of 800 px or more, the stroke won't be violet.

Conversely, when you insert an SVG inline, it merges (in a manner of speaking) with the outer HTML document. The `min-width` media query here is looking to the viewport (as is the HTML) to decide when the media query matches.

To solve this particular problem and make the same media query behave consistently, we could amend our media query to this:

```
@media (min-device-width: 800px) {
  #star_Path {
    stroke: violet;
  }
}
```

That way, regardless of the SVG size or how it is embedded, it looks to the device width (effectively the viewport).

Optimizing SVGs

As conscientious developers, we want to ensure that assets are as small as possible. The easiest way to do this with SVGs is to make use of automation tools that can optimize various particulars of SVG documents. Besides obvious economies such as removing elements (for example, stripping the title and description elements), it's also possible to perform a raft of micro-optimizations that, when added up, make for far leaner SVG assets. Presently, for this task, I would recommend SVGO (`https://github.com/svg/svgo`).

If you have never used SVGO before, I would recommend starting with SVGOMG (`https://jakearchibald.github.io/svgomg/`). It's a browser-based version of SVGO that enables you to toggle the various optimization plugins and get instant feedback on the file savings. Remember our example star SVG markup from the beginning of this chapter? By default, that simple SVG is 489 bytes in size. By passing that through SVGO, it's possible to get the size down to just 218 bytes, and that's leaving `viewBox` in. That's a saving of 55.42%. If you're using a raft of SVG images, these savings can really add up. Here's what the optimized SVG markup looks like:

```
<svg
  width="198"
  height="188"
  viewBox="0 0 198 188"
  xmlns="http://www.w3.org/2000/svg"
>
  <path
    stroke="#979797"
    stroke-width="3"
    fill="#F8E81C"
    d="M99 154l-58.78 30.902 11.227-65.45L3.894 73.097l65.717-9.55L99
      4l29.39 59.55 65.716 9.548-47.553 46.353 11.226 65.454z"
  />
</svg>
```

Before you spend too long with SVGO, be aware that, such is the popularity of SVGO, plenty of other SVG tools also make use of it. For example, the aforementioned iconizr (`https://iconizr.com`) tool runs your SVG files through SVGO by default anyway, before creating your assets, so ensure you aren't unnecessarily double optimizing.

SVG implementation tips

We're almost at the end of this chapter now, but there is still so much we could talk about regarding SVG. Therefore, at this point, I'll just list a few unrelated considerations. They aren't necessarily worthy of protracted explanations, but I'll list them here in note form to save you from hours of searching Stack Overflow or asking your AI chatbot of choice:

- If you have no need to animate your SVGs, opt for an image sprite of your assets or use data URIs in your style sheet.
- Automate as many steps in the asset creation process as possible; this reduces human error and produces predictable results faster.
- To insert static SVGs in a project, pick a single delivery mechanism and stick to it (image sprite, data URI, or inline `mask-image`). It can become a burden to produce some assets one way and some another and maintain the various implementations.
- There is no easy "one size fits all" choice with SVG animation. For occasional and simple animations, use CSS. For complex interactive or timeline-style animations, lean on a proven library such as GSAP. Or, look at using the JavaScript Web Animations API.

These are just guides, not gospel. There will be times when you need to make a choice that may seem counterintuitive to conventional wisdom. But at least if you have taken in the details here, you will be able to make informed choices when it comes to working with SVG.

Summary

This has been a dense chapter. We have covered a lot of the essential information needed to start making sense of and implementing SVGs in a responsive project. We have considered the different graphics applications and online solutions available to create SVG assets, then the various insertion methods possible and the capabilities each allows, along with the various browser peculiarities to be aware of.

We've also considered how to link to external style sheets and reuse SVG symbols from within the same page and when referenced externally. We even looked at how we can make filters with SVG that can be referenced and used in CSS.

Finally, we considered how to make use of JavaScript libraries to aid in animating SVGs, as well as how to optimize SVGs with the aid of the SVGO tool.

In the next chapter, we'll be looking at CSS transitions, transforms, and animations. It's also worth reading that chapter in relation to SVG, as many of the syntaxes and techniques can be used and applied in SVG documents too.

Further resources

As I mentioned at the start of this chapter, I have neither the space nor the knowledge to impart all there is to know about SVG. Therefore, I'd like to make you aware of the following excellent resources, which provide additional depth and range on the subject:

- *SVG Essentials, 2nd Edition*, by J. David Eisenberg and Amelia Bellamy-Royds: `https://shop.oreilly.com/product/0636920032335.do`
- *A Guide to SVG Animations (SMIL)*, by Sara Soueidan: `https://css-tricks.com/guide-svg-animations-smil/`
- Testing media queries inside SVGs, by Jeremie Patonnier: `http://jeremie.patonnier.net/experiences/svg/media-queries/test.html`
- *An SVG Primer for Today's Browsers*: `https://www.w3.org/Graphics/SVG/IG/resources/svgprimer.html`
- *Understanding SVG Coordinate Systems and Transformations (Part 1)*, by Sara Soueidan: `https://sarasoueidan.com/blog/svg-coordinate-systems/`
- *Hands On: SVG Filter Effects*: `https://testdrive-archive.azurewebsites.net/Graphics/hands-on-css3/hands-on_svg-filter-effects.htm`
- The full set of SVG tutorials by Jakob Jenkov: `https://jenkov.com/tutorials/svg/index.html`

12

Transitions, Transformations, and Animations

CSS can handle the majority of motion requirements using CSS transitions and transforms, or CSS animations.

Besides often being aesthetically pleasing, movement in a user interface, used judiciously, can help to *join up* one state to another, communicating a journey of sorts and therefore aiding a user in completing their task.

To clearly understand what transitions, transforms, and animations do, I will offer this, perhaps overly simplistic, summary:

- A CSS **transition** is used to define how one visual state should move (transition) to another, differing visual state.
- A CSS **transform** is used to take an existing element and transform it into something or someplace else without affecting any other elements on the page. For example, "make this twice as big" and "move this 100px to the right" are plain text descriptions of tasks we can achieve with CSS transforms. However, the transform doesn't control *how* the element makes that change; that is the job of the transition.
- A CSS **animation** is typically used to make a series of changes to an element.

If those differences seem a little vague at this point, hopefully, by the end of this chapter, they won't.

In this chapter, we'll cover the following:

- What CSS transitions are and how to use them
- Writing CSS transitions and understanding the shorthand syntax
- CSS transition timing functions (`ease`, `cubic-bezier`, and so on)
- What CSS transforms are and how to use them
- Understanding different 2D transforms (`scale`, `rotate`, `skew`, `translate`, and `matrix`)
- Understanding 3D transforms
- How to animate with CSS using keyframes
- Using `@starting-style` to transition elements into view
- Transition elements to hidden with `transition-behavior`
- Animating height changes with `calc-size()`

Let's get started with an introduction to CSS transitions.

What CSS transitions are and how we can use them

Transitions are the simplest way to create some visual "effect" between one state and another with CSS.

Let's consider a simple example: an element that transitions from one state to another when hovered over.

When styling hyperlinks in CSS, it's common practice to create a `hover` state: an obvious way to make users aware that the item they are hovering over can be interacted with. Hover states are of little relevance to the growing number of touchscreen devices, but for mouse users, they're a great and simple interaction between the website and the user. They're also handy for illustrating transitions, so that's what we will start with.

Traditionally, using only CSS, `hover` states are an on/off affair. There is one set of properties and values on an element as the default, and when a pointer is hovered over that element, the properties and values are instantly changed. However, CSS transitions, as the name implies, allow us to transition between one or more properties and values to other properties and values.

A couple of important things to know upfront. Firstly, until very recently, you have not been able to transition from `display: none`. When something is set to `display: none`, it isn't actually "painted" on the screen, so in all but the latest browsers, it has no existing state you can transition from. I'll show you how we can do this simply in newer browsers shortly.

Although you can't transition from `display: none`, it is possible to run an animation on an element as soon as its `display` property is changed. So, you could, for example, change an element's `display` property from `display: none` and then run an animation to fade it in from 0% opacity. We'll cover animations later in this chapter.

In order to create the effect of something fading in, you would have to transition opacity or position values. Secondly, not all properties can be transitioned. To ensure you're not attempting the impossible, here is the list of transitionable (I know, it's not even a word) properties: `https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_animated_properties`.

If you open up `example_12-01`, you'll see a few links in a `nav`. Here's the relevant markup:

```
<nav>
  <a href="#">link1</a>
  <a href="#">link2</a>
  <a href="#">link3</a>
  <a href="#">link4</a>
  <a href="#">link5</a>
</nav>
```

And here's the relevant CSS:

```
a {
  font-family: sans-serif;
  color: #fff;
  text-indent: 1rem;
  background-color: #ccc;
  display: inline-flex;
  flex: 1 1 20%;
  align-self: stretch;
  align-items: center;
  text-decoration: none;
  transition: box-shadow 1s;
}
a + a {
  border-left: 1px solid #aaa;
}
a:hover {
  box-shadow: inset 0 -3px 0 #cc3232;
}
```

And here are the two states. First, the default:

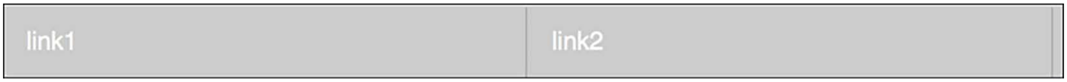


Figure 12.1: Two links, in their default state

And then here's the `hover` state:

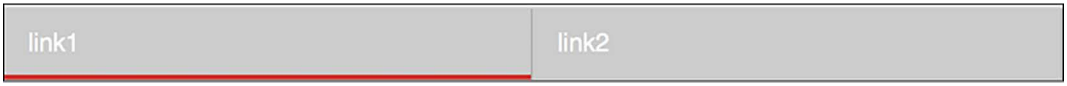


Figure 12.2: When hovered, we underline the link

Ordinarily, hovering over the link snaps from the first state (no red line) to the second (red line); it's an on/off affair. However, the following line adds a transition to `box-shadow` from the existing state to the `hover` state over 1 second:

```
transition: box-shadow 1s;
```

> You'll notice in the CSS of the preceding example that we're using the adjacent sibling selector, +. You'll remember this selector from *Chapter 6*, *CSS Nesting, Layers, Selectors, and More*. This means if an element (an anchor tag in our example) directly follows another element (another anchor tag), then apply the enclosed styles. It's useful here as we don't want a left border on the first element.

The `transition` property is applied to the "from" state, not the "to" state of a selector. In our case, we add the `transition` property and values on the element in its default state, not the `hover` state. You could add a `transition` property to the `hover` state, too, but this would apply when you transition away from the `hover` state and back to the default. This implementation is so that multiple different states, such as `:active`, `:focus`, `:hover`, and more, can also have different styles set and enjoy the same transition to them.

The properties of a transition

A transition can be declared using up to four properties:

- `transition-property`: The name of the CSS property to be transitioned (such as `background-color`, `text-shadow`, or `all` to transition every possible property).
- `transition-duration`: The length of time over which the transition should occur (defined in seconds, for example, `.3s`, `2s`, or `1.5s`).

- `transition-timing-function`: How the transition changes speed during the duration (for example, `ease`, `linear`, `ease-in`, `ease-out`, `ease-in-out`, or `cubic-bezier`).
- `transition-delay`: An optional value to determine a delay before the transition commences. Alternatively, a negative value can be used to commence a transition immediately, but partway through its transition "journey." It's defined in seconds, for example, `.3s`, `1s`, or `2.5s`.

> For any duration defined in seconds with `s` in CSS, you can also define it in milliseconds, if you prefer. For example, instead of `0.5s` for half a second, you could write `500ms`. It's purely a preference thing, but for predictability, I would advise you to choose one or the other and stick to it throughout a project.

Used separately, the various transition properties can be used to create a transition like this:

```
.style {
  /*...(more styles)...*/
  transition-property: all;
  transition-duration: 1s;
  transition-timing-function: ease;
  transition-delay: 0s;
}
```

This particular rule effectively says, "Transition any transitionable property on this element over 1 second with no delay using the `ease` timing function."

The transition shorthand property

We can roll these individual declarations into a single, shorthand version:

```
transition: all 1s ease 0s;
```

One important point to note when writing the shorthand version is that the first time-related value provided is always parsed to be the `transition-duration`. The second time-related value is parsed to be `transition-delay`. The shorthand version is the one I favor, as I generally only need to define the duration of the transition and the properties that should be transitioned.

To exemplify, suppose I wanted the `background-color` to transition over 2 seconds:

```
transition: background-color 2s;
```

If no timing function is provided, the default timing function of `ease` is applied.

I would also advise only defining the property or properties you actually need to transition. It's really handy to just set `all`, but, for example, if you only need to transition `opacity`, then only define `opacity` as the transition property. When you set `all`, there is a chance you will be inadvertently transitioning properties you don't want to, which can possibly lead to bugs for yourself or others down the line.

Transitioning different properties over different periods of time

Where a rule has multiple properties declared, you don't have to transition all of them in the same way. Consider this rule:

```
.style {
  /* ...(more styles)... */
  transition-property: border, color, text-shadow;
  transition-duration: 2s, 3s, 8s;
}
```

Here, we have specified with `transition-property` that we'd like to transition `border`, `color`, and `text-shadow`. Then, with the `transition-duration` declaration, we are stating that `border` should transition over 2 seconds, `color` over 3 seconds, and `text-shadow` over 8 seconds. The comma-separated durations match the comma-separated order of the transition properties.

You can also perform the same job with the shorthand syntax, and again, this is my preferred way to do it:

```
.style {
  transition: border 2s, color 3s, text-shadow 8s;
}
```

I think it is far easier to reason about each property and its timing when they are written next to each other.

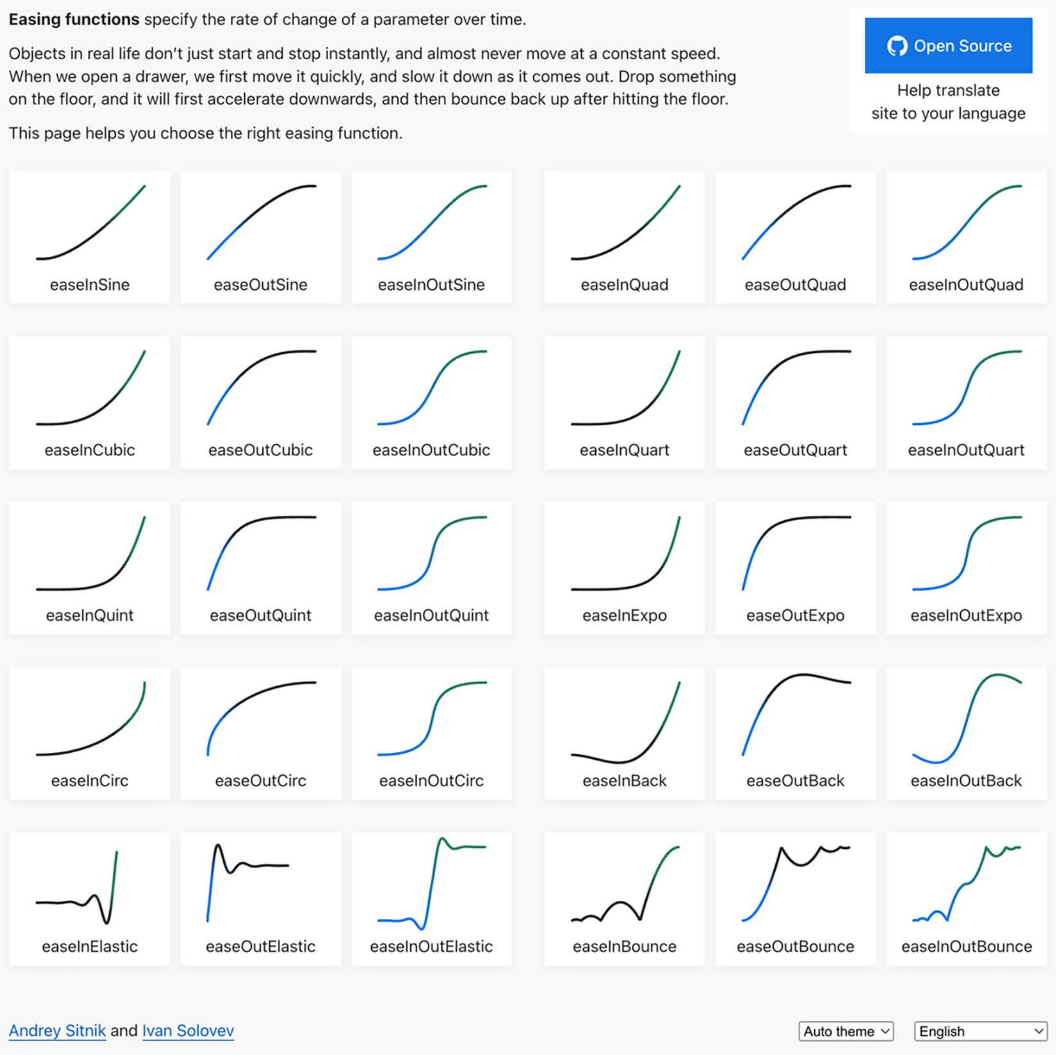
So far, we have dealt with transitioning different properties over different durations. However, there is also a fairly limitless number of timing functions that change how an element property changes over a duration.

Understanding timing functions

When you declare a transition, the properties, durations, and delays are relatively simple to understand. However, understanding what each timing function does can be a little trickier. Just what do `ease`, `linear`, `ease-in`, `ease-out`, `ease-in-out`, and `cubic-bezier` actually do?

Each of them is actually a predefined `cubic-bezier` curve, essentially the same as an easing function. Or, more simply, a mathematical description of how the transition should look. It's generally easier to visualize these curves, so I recommend you head over to `https://cubic-bezier.com/` or `https://easings.net`.

Both these sites let you compare timing functions and see the difference each one makes. Here is a screenshot of `https://easings.net`—on the website, you can hover over each line for a demonstration of the easing function:



Easing functions specify the rate of change of a parameter over time.

Objects in real life don't just start and stop instantly, and almost never move at a constant speed. When we open a drawer, we first move it quickly, and slow it down as it comes out. Drop something on the floor, and it will first accelerate downwards, and then bounce back up after hitting the floor.

This page helps you choose the right easing function.

Help translate site to your language

Andrey Sitnik and Ivan Solovev

Open Source
easeInSine
easeOutSine
easeInOutSine
easeInQuad
easeOutQuad
easeInOutQuad
easeInCubic
easeOutCubic
easeInOutCubic
easeInQuart
easeOutQuart
easeInOutQuart
easeInQuint
easeOutQuint
easeInOutQuint
easeInExpo
easeOutExpo
easeInOutExpo
easeInCirc
easeOutCirc
easeInOutCirc
easeInBack
easeOutBack
easeInOutBack
easeInElastic
easeOutElastic
easeInOutElastic
easeInBounce
easeOutBounce
easeInOutBounce
Auto theme
English

Figure 12.3: Easings.net provides a handy visualization of a number of different timing functions

However, even if you can write your own `cubic-bezier` curves blindfolded, the likelihood is that, for most practical situations, it makes little difference. The reason for this is that, like any visual flourish, it's necessary to employ transition effects subtly. For real-world implementations, transitions that occur over too great a period of time tend to make a site feel slow. For example, navigation links that take 5 seconds to transition are going to frustrate rather than wow your users. The perception of speed is incredibly important for our users, and you and I must concentrate on making websites and applications feel as fast as possible.

Therefore, unless there is a compelling reason to do so, using the default transition (`ease`) over a short interval is often best; a maximum of `1s` is my own preference, and usually as little as `0.2s`.

Fun transitions for responsive websites

Did you ever have one of those occasions growing up when one parent was out for the day and the other parent said something to the effect of, "OK, while your mom/dad is out, we're going to put sugar all over your breakfast cereal, but you have to promise not to tell them when they come back." I'm certainly guilty of that with my little ankle biters. So, here's the thing. While no one is looking, let's have a bit of fun. I don't recommend this for production, but try adding this to your responsive project:

```
* {
  transition: all 1s;
}
```

Here, we are using the * CSS universal selector to select everything and then setting a transition on all properties for 1 second (`1s`). As we have omitted to specify the timing function, `ease` will be used by default, and there will be no delay as, again, a default of `0` is assumed if an alternative value is not added. The effect? Well, try resizing your browser window, and most things (links, `hover` states, and the like) behave as you would expect. However, because everything transitions, it also includes any rules within media queries, so as the browser window is resized, elements sort of flow from one state to the next. Is it essential? Absolutely not! Is it fun to watch and play around with? Certainly! Now, remove that rule before your mom sees it!

Right, hopefully, we have transitions licked. Now we can have some fun, because we are going to learn how to effortlessly move elements around the screen with transforms. There are 2D and 3D variants; we will start with 2D, and if that gets a bit flat (see what I did there? No? I'll see myself out), then we will move on to 3D.

CSS 2D transforms

Despite sounding similar to CSS transitions, CSS transforms are entirely different. As we already established, transitions deal with the transition from one state to another. Transforms, on the other hand, are a means of defining what the state should actually be.

My own (admittedly childish) way of remembering the difference is like this: imagine a Transformer robot such as Optimus Prime. When he has changed into a truck, he has transformed. However, the period between robot and truck is a transition (he's transitioning from one state to another). Obviously, if you have no idea who or what Optimus Prime even is, feel free to mentally discard the last few sentences. Hopefully, all will become clear when we get to the examples in a moment.

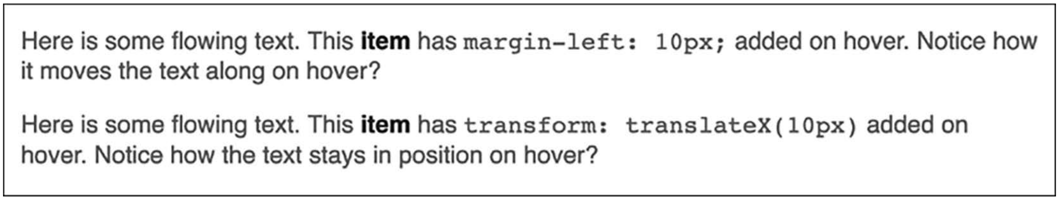
There are two groups of CSS transforms available: 2D and 3D. 2D variants are probably a little more straightforward, so let's look at those first. The CSS 2D Transforms Module allows us to use the following transforms:

- `scale`: Scale an element (larger or smaller)
- `translate`: Move an element on the screen (up, down, left, and right)
- `rotate`: Rotate the element by a specified amount (defined in degrees or turns)
- `skew`: Skew an element with its *X* and *Y* coordinates
- `matrix`: Move and shape transformations in multiple ways

An essential concept to understand is that transforms occur *outside of the document flow*. More simply, any element that is transformed will not affect the position of any other element that is not a child of it. This is quite different from adding a transition to an element when you change the margin, height, or other transitionable/animatable property.

Open `example_12-02` and you will see two sentences; the first has `margin-left: 10px` applied to the bold word **item** on hover, while the second sentence has `transform: translateX(10px)` applied to the same word on hover. Both have a transition of 1 second applied.

With nothing hovering over, they look like this:

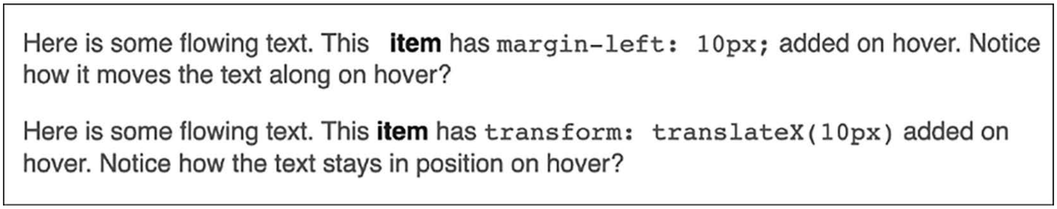


Here is some flowing text. This **item** has `margin-left: 10px;` added on hover. Notice how it moves the text along on hover?

Here is some flowing text. This **item** has `transform: translateX(10px)` added on hover. Notice how the text stays in position on hover?

Figure 12.4: Two paragraphs in their default state

Hover over **item** in the first paragraph, and the following happens: notice the text after that word has been shunted along?

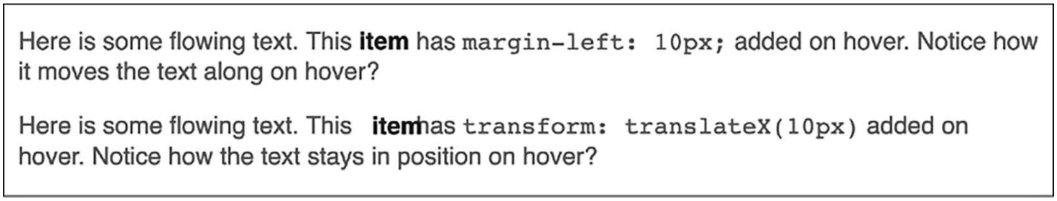


Here is some flowing text. This **item** has `margin-left: 10px;` added on hover. Notice how it moves the text along on hover?

Here is some flowing text. This **item** has `transform: translateX(10px)` added on hover. Notice how the text stays in position on hover?

Figure 12.5: When the first paragraph is hovered over, "item" gets a margin applied, moving all the text after it too

Now, hover over **item** in the second paragraph and notice how just that word moves. The words around it are unaffected.



Here is some flowing text. This **item** has `margin-left: 10px;` added on hover. Notice how it moves the text along on hover?

Here is some flowing text. This **item**has `transform: translateX(10px)` added on hover. Notice how the text stays in position on hover?

Figure 12.6: In the second paragraph, "item" is transformed, moving to the right without affecting anything else

So, transforms, either 2D or 3D, do not affect document flow.

Let's try out the various 2D transforms. You can test each of these out by opening `example_12-03` in the browser. There's a transition applied to all of the transforms so that you get a better idea of what's happening.

Scale

Here's the syntax for `scale`:

```
.scale:hover {
  transform: scale(1.4);
}
```

Hovering over the **scale** link in our example produces this effect:

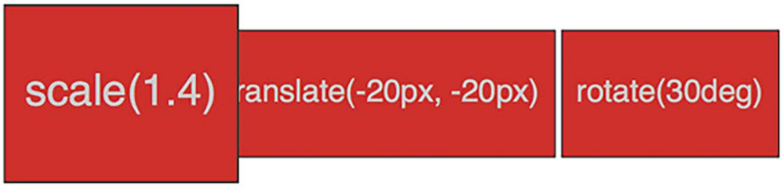


Figure 12.7: You can scale elements up or down in size

We've told the browser that when this element is hovered over, we want the element to scale to 1.4 times its original value.

Besides the values we've already used to enlarge elements, by using values below `1`, we can shrink elements; the following will shrink the element to half its size:

```
transform: scale(0.5);
```

You can also apply a scale only on a single axis. Horizontal is `scaleX`, and vertical is `scaleY`. So, if you wanted to scale something to twice its width, you'd use the following:

```
transform: scaleX(2);
```

Translate

Here's the syntax for `translate`:

```
.translate:hover {
  transform: translate(-20px, -20px);
}
```

Here's the effect that rule has on our example:

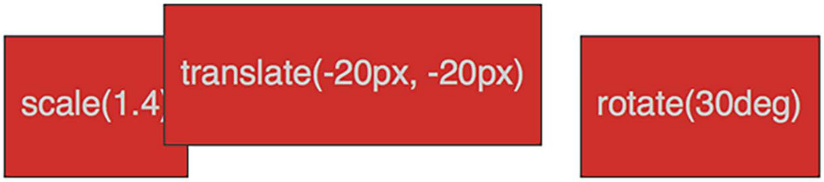


Figure 12.8: translate lets you move an element anywhere on the x or y axis

The `translate` property tells the browser to move an element by an amount, defined by a length (for example, vw, px, %, and so on). The first value is the *x* axis, and the second value is the *y* axis. Positive values given within parentheses move the element right or down; negative values move it left or up. If you only pass one value, then it is applied to the *x* axis.

If you want to specify just one axis to translate an element, you can also use `translateX(-20px)`, which, in this instance, would move the element left 20 px, or you could use `translateY(-20px)`, which, in this case, would move the element 20 px up.

Using translate to center absolutely positioned elements

`translate` provides a really useful way to center absolutely positioned elements within a relatively positioned container. You can view this example at `example_12-04`.

Consider this markup:

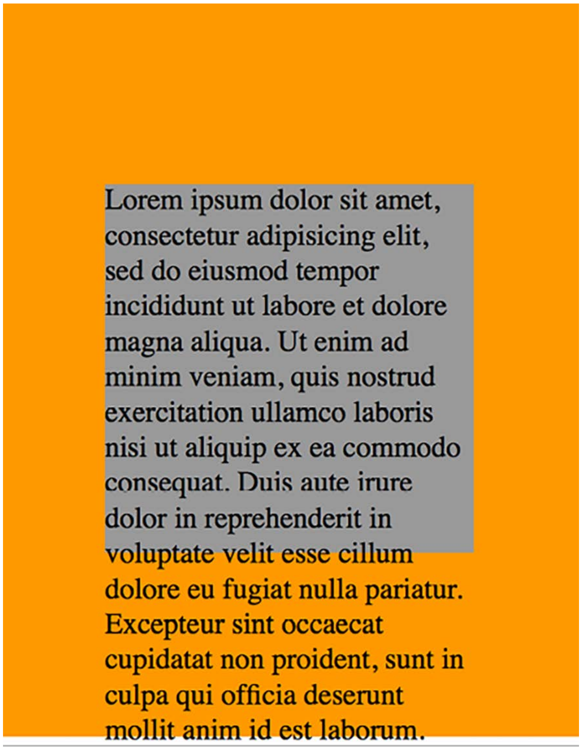
```
<div class="outer">
  <div class="inner"></div>
</div>
```

And then consider this CSS:

```
.outer {
  position: relative;
  height: 400px;
  background-color: #f90;
}
.inner {
  position: absolute;
  height: 200px;
  width: 200px;
  margin-top: -100px;
  margin-left: -100px;
  top: 50%;
  left: 50%;
}
```

You've perhaps done something similar to this yourself. When the dimensions of the absolutely positioned element are known (200 px × 200 px, in this case), we can use negative margins to "pull" the item back to the center. However, what happens when you want to include content and have no way of knowing how tall it will be?

For example, let's add some random content to the inner box:



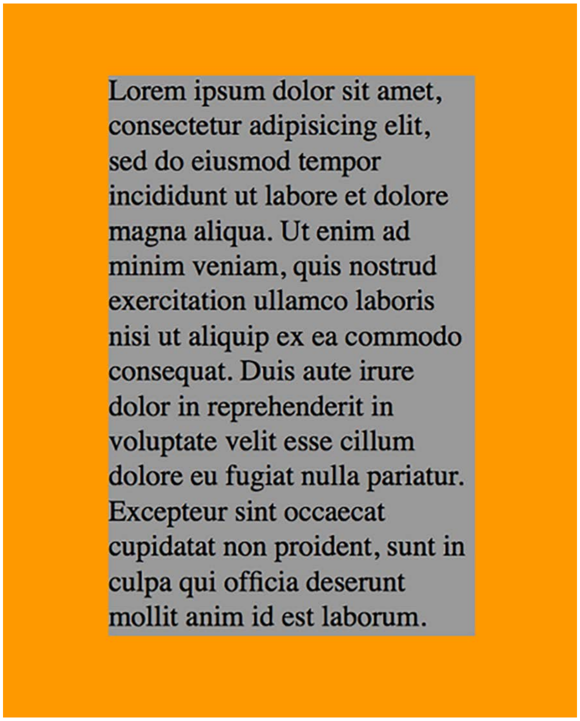
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 12.9: Text overflowing our box can be fixed with transform

Yes, that problem! Right, let's use `transform` to sort this mess out:

```
.inner {
  position: absolute;
  width: 200px;
  background-color: #999;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);
}
```

And here is the result:



Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 12.10: With a smart application of transform, overflow issues are avoided

Here, `top` and `left` are positioning the inner box inside its container so that the top-left corner of the inner box starts at a point 50% along and 50% down the outer box. Then, `transform` is working on the inner element and positioning it negatively in those axes by half (`-50%`) of its own width and height. Nice!

> One aspect of what we learned here bears repeating. When using percentages in transforms, remember that they are a percentage of *the element being transformed*, not its parent.

Rotate

The `rotate` transform allows you to rotate an element. Here's the syntax:

```
.rotate:hover {
  transform: rotate(30deg);
}
```

In the browser, here's what happens:

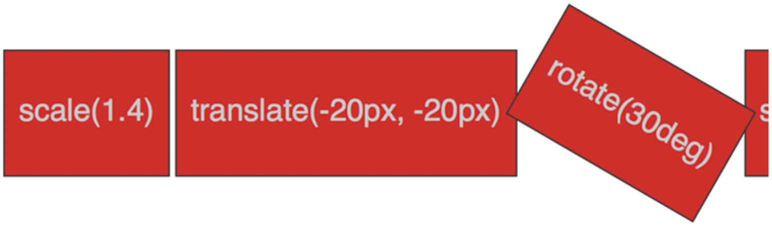


Figure 12.11: Rotating elements is straightforward with a transform

The value in parentheses should always be an angle. The angle can be expressed in degrees, gradians, radians, or turns. Personally, I default to using degrees (for example, `90deg`), but all units are equally valid. While positive angle values always apply clockwise, using negative values will rotate the element counterclockwise.

Pass an angle greater than a full revolution, and the element will keep turning until it has turned around to the required degree. Therefore, you can also go crazy and make elements spin by specifying a value such as the following:

```
transform: rotate(3600deg);
```

This will rotate the element 10 times in a complete circle. Practical uses for this particular value are few and far between, but you know, if you ever find yourself designing websites for a windmill company, it may come in handy.

Skew

If you've spent any time working in Photoshop, you'll have a good idea of what `skew` does. It allows an element to be skewed on either or both of its axes. Here's the code for our example:

```
.skew:hover {
  transform: skew(40deg, 12deg);
}
```

Setting this on the `hover` state produces the following effect on hover:

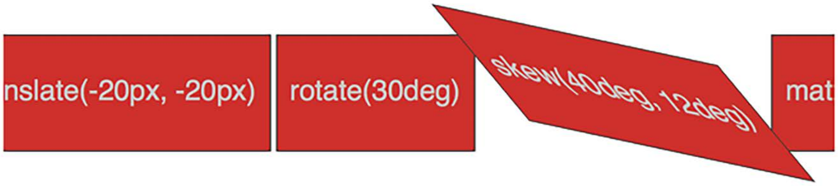


Figure 12.12: Just like it sounds, a skew can produce some dramatic effects

The first value is `skew` applied to the *x* axis (in our example, `40deg`), while the second (`12deg`) is for the *y* axis. Omitting the second value means any value will merely be applied to the *x* axis (horizontal), as in this example:

```
transform: skew(10deg);
```

Just like with `translate`, you can apply skews to just one axis with `skewX()` and `skewY()`.

Matrix

Did somebody mention a completely overrated film? No? What's that? You want to know about the CSS `matrix` transform, not the film? Oh, okay.

The `matrix` transform syntax looks rather impenetrable. Here's our example code:

```
.matrix:hover {
  transform: matrix(1.178, -0.256, 1.122, 1.333, -41.533, -1.989);
}
```

The matrix is a mathematical description. It essentially allows you to combine a number of other transforms (`scale`, `rotate`, `skew`, and so on) and express them as a single declaration. The preceding declaration results in the following effect in the browser:

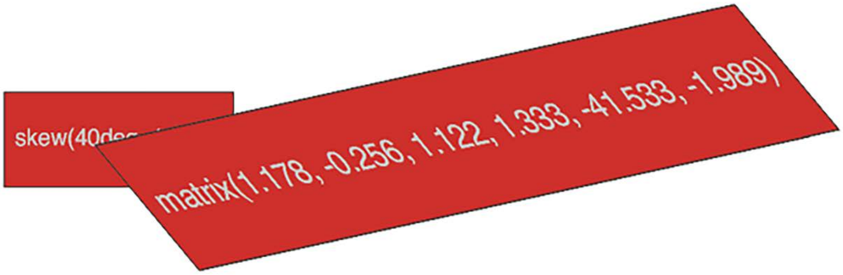


Figure 12.13: Not for the faint-hearted, the matrix value

If you find yourself doing work with animations in JavaScript without the help of an animation library, you'll probably need to become a little more acquainted with `matrix`. It's the syntax all the other transforms get computed into behind the scenes, so if you're grabbing the current state of an animation with JavaScript, it will be the `matrix` value you will need to inspect and understand.

Now, I like a challenge (unless, you know, it's sitting through the Marvel MCU back catalog), but I think the majority of us would agree that that syntax is a bit testing. The specification doesn't exactly clear matters up: `https://www.w3.org/TR/css-transforms-1/#mathematical-description`.

However, the truth is, I can count on the fingers of one hand how many times I've needed to write or understand a CSS transform described as `matrix`, so it likely isn't something to concern yourself with.

If you do find yourself needing to create one, I'd suggest heading over to `https://www.useragentman.com/matrix/`. The Matrix Construction Set website allows you to drag and drop the element exactly where you want it and includes good ol' copy-and-paste code for your CSS file.

The transform-origin property

Notice how, with CSS, the default transform origin—the point the browser uses as the center for the transform—is in the middle: 50% along the *x* axis and 50% along the *y* axis of the element. This differs from SVG, which defaults to the top left (or `0 0`).

Using the `transform-origin` property, we can amend the point from which transforms originate.

Consider our earlier `matrix` transform. The default `transform-origin` is `50% 50%` (the center of the element). The Firefox developer tools show how the transform is applied:

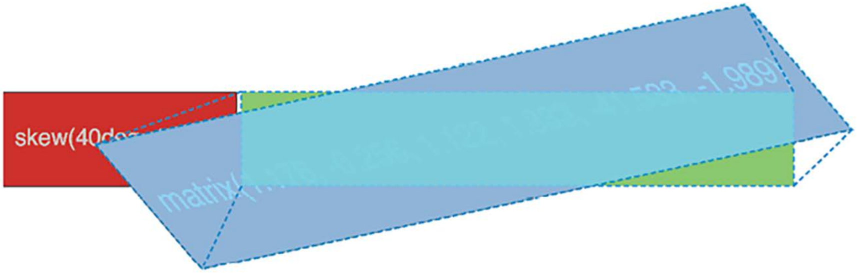


Figure 12.14: Default transform-origin applied

Now, let's adjust `transform-origin` like this:

```
.matrix:hover {
  transform: matrix(1.678, -0.256, 1.522, 2.333, -51.533, -1.989);
  transform-origin: 270px 20px;
}
```

You can see the effect this has:

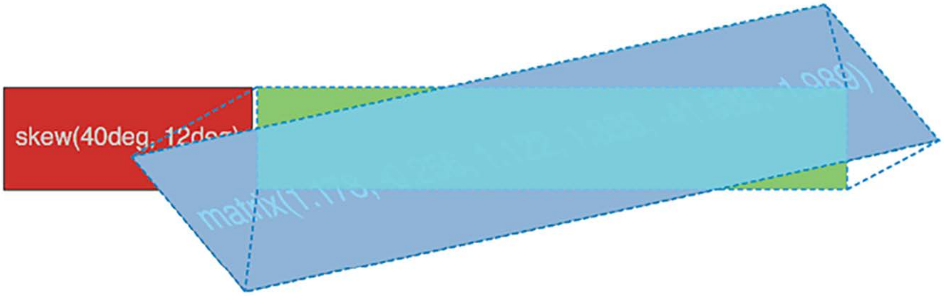


Figure 12.15: You can amend the origin of transforms as needed

The first value is the horizontal offset, and the second value is the vertical offset. You can use keywords. For example, `left` is equal to 0% horizontal, `right` is equal to 100% horizontal, `top` is equal to 0% vertical, and `bottom` is equal to 100% vertical. Alternatively, you can use a length, using any of the CSS length units.

If you use a percentage for the `transform-origin` values, then the horizontal/vertical offset is relative to the height/width of the element's bounding box. If you use a length, then the values are measured from the top-left corner of the element's bounding box.

The specification for CSS Transforms Module Level 1 can be found here: `https://www.w3.org/TR/css-transforms-1`.

> For more on the benefits of moving elements with `transform`, read this old but great post by Paul Irish, `https://www.paulirish.com/2012/why-moving-elements-with-translate-is-better-than-posabs-topleft/`, which provides some good data.

So far, we have dealt with transforming in two dimensions, the *x* and *y* axes. However, CSS can also handle elements in 3D space. Let's look at what extra fun we can have with 3D transforms.

CSS 3D transformations

As you've probably already realized, a 3D transform allows us to manipulate an element in an imaginary 3D space. Let's look at our first example. All we have in our example are two elements that each flip in 3D when hovered over. I've used `hover` here to invoke the flip, as it's a simple mechanism to trigger the effect. However, the flipping action could just as easily be initiated with any other state change—a class change (via JavaScript) or when an element has received focus, for example.

The only difference between these two elements is that one flips horizontally and the other vertically. You can view them in a browser by opening `example_12-05`. Images fail to fully convey this technique, but the idea is that the element flips from the green "face" to the red "face," giving the illusion of doing so through 3D space with the aid of perspective. Here's a grab partway through the transition from green to red, which hopefully conveys some of the effect:

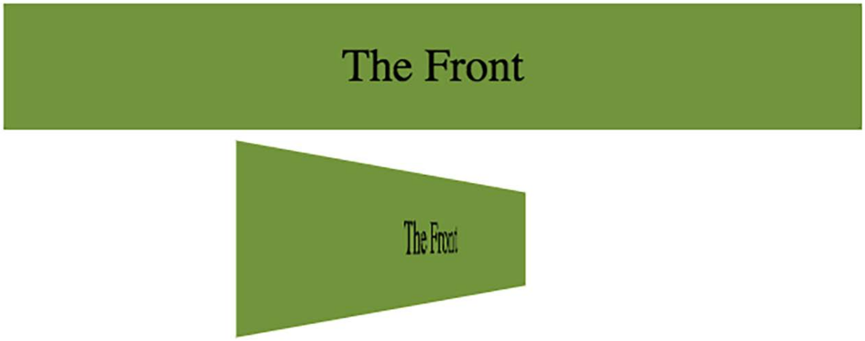


Figure 12.16: Halfway through a 3D transform

It's also worth knowing that while positioning an element absolutely with `top`/`left`/`bottom`/`right` values can only be calculated to whole pixels, a transform can interpolate at subpixel positions.

Here's the markup for the flipping element:

```
<div class="flipper">
  <span class="flipper-object flipper-vertical">
    <span class="panel front">The Front</span>
    <span class="panel back">The Back</span>
  </span>
</div>
```

The only difference with the horizontal one, markup-wise, is that the `flipper-vertical` class is replaced with `flipper-horizontal`.

As the majority of the styles relate to aesthetics, we'll merely look at the essential ingredients in our styles to make the flipping effect possible. Refer to the full style sheet in the example file for the aesthetic styles.

When dealing with 3D transforms, the first essential ingredient is to define some kind of 3D space.

For our example, we need to set some perspective for `.flipper-object`, whether horizontal or vertical, to flip within. For that, we use the `perspective` property. This takes a length value, attempting to simulate the distance from the viewer's screen to the edge of the elements' 3D space. We set this on the outermost element to provide the 3D context for our nested elements to move within.

The value you assign to `perspective` is arguably counter-intuitive—if you set a low number such as `20px` for the `perspective` value, the 3D space of the element will extend right out to only 20 px from your screen, the result being a more pronounced 3D effect.

Setting a high number, on the other hand, will mean the edge of that imaginary 3D space will be further away, and therefore, produce a less pronounced 3D effect. I recommend opening the example in the browser and playing with the perspective in the developer tools to get a feel for it:

```
.flipper {
  perspective: 400px;
  position: relative;
}
```

We use `position: relative` on the outer `flipper` element to provide a positioning context for `flipper-object` to be absolutely positioned within:

```
.flipper-object {
  position: absolute;
  width: 100%;
  height: 100%;
  transition: transform 1s;
  transform-style: preserve-3d;
}
```

Besides positioning `.flipper-object` absolutely, we set the `height` and `width` values to `100%` so that it fills the same space as the outer container.

We have also set a transition for the transform. If you remember from earlier in this chapter, when we dealt with transitions, just setting the time for transition means the default timing function (`ease`) will be used.

The key thing here, 3D-wise, is `transform-style: preserve-3d`. This declaration tells the browser that when we transform this element, we want any child elements to preserve any 3D effect.

If we didn't set `preserve-3d` on `.flipper-object`, we would never get to see the back (the red part) of the flipping element.

You can read the specification for this property here: `https://www.w3.org/TR/2009/WD-css3-3d-transforms-20090320/#transform-style-property`.

Each "panel" in our flipping element needs positioning at the top of its container, but we also want to make sure that, if rotated, we don't see the "rear" of it (otherwise we would never see the green panel as it sits "behind" the red one). To do that, we use the `backface-visibility` property. We set this to `hidden` so that the back face of the element is, you guessed it, hidden:

```
.panel {
  top: 0;
  position: absolute;
  backface-visibility: hidden;
}
```

Next, we want to make our back panel flipped by default (so that when we flip the whole thing, it will actually be in the correct position). To do that, we apply a `rotate` transform. Hopefully, having covered these in the previous section, you'll understand what they are doing here:

```
.flipper-vertical .back {
  transform: rotateX(180deg);
}
.flipper-horizontal .back {
  transform: rotateY(180deg);
}
```

Now that everything is in place, all we want to do is flip the entire inner element when the outer one is hovered over:

```
.flipper:hover .flipper-vertical {
  transform: rotateX(180deg);
}
```

```
.flipper:hover .flipper-horizontal {
  transform: rotateY(180deg);
}
```

As you can imagine, there are a bazillion (by the way, bazillion is definitely not a real amount, I just checked) ways you can use these techniques. If you're wondering what a fancy navigation effect, or off-canvas menu, might look like with a spot of perspective, I highly recommend paying Codrops a visit: `https://tympanus.net/Development/PerspectivePageViewNavigation/index.html`.

You can read the W3C specification for the CSS Transforms Module Level 1 at `https://www.w3.org/TR/css-transforms-1/`.

It turns out there is a very handy property when working with transforms that can do all your *x*, *y*, and *z* movement in one go. Let's take a look at that next.

The translate3d property

I've found great utility in the `translate3d` function. With this single function, it is possible to move an element in the *x* (left/right), *y* (up/down), and *z* (forward/backward) axes.

Let's amend our previous example and make use of the `translate3d()` function. You can view this example at `example_12-06`.

Besides setting the elements into the page a little more with padding, the only changes from our previous example can be seen here:

```
.flipper:hover .flipper-vertical {
  transform: rotateX(180deg) translate3d(0, 0, -120px);
}
.flipper:hover .flipper-horizontal {
  transform: rotateY(180deg) translate3d(0, 0, 120px);
}
```

We're still applying a transform, but this time, in addition to `rotate`, we have also made use of `translate3d()`. The syntax for the comma-separated arguments you can pass into `translate3d` are *x*-axis movement, *y*-axis movement, and *z*-axis movement.

In our two examples, I'm not moving the element on the *x* or *y* axis (left to right, and up and down); instead, I'm moving it toward or further away from you as you look at it.

If you look at the top example, you will see it flip behind the bottom button and end 120 px closer to the screen (use minus values to effectively pull it back toward you):

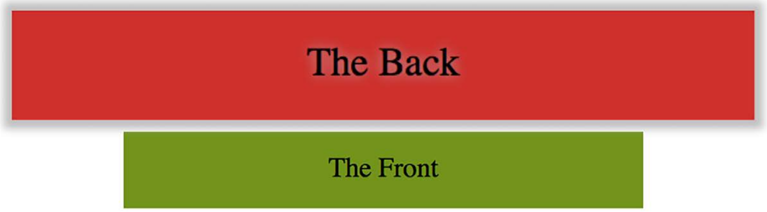


Figure 12.17: Elements can be moved forward and backward in the Z plane

On the other hand, the bottom button flips around horizontally and ends with the button 120 px farther away from you:

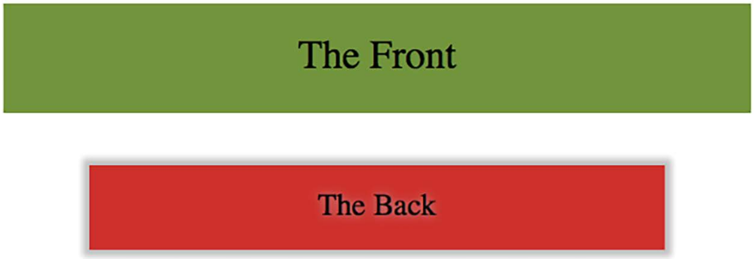


Figure 12.18: Here, the element ends farther away

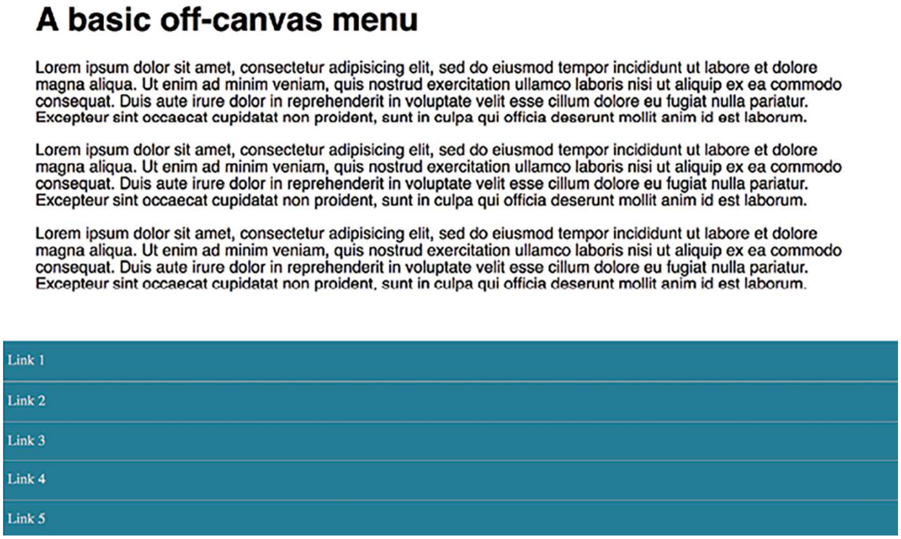
Although `translate3d()`, like the other transforms, is very well supported, it actually got pushed into the later CSS Transforms Module Level 2 specification. You can read about that here: `https://www.w3.org/TR/css-transforms-2/`.

A progressive enhancement example using translate3d

An area I have found using `translate3d` useful in is sliding panels on and off the screen, particularly "off-canvas" navigation patterns. If you open `example_12-07`, you'll see I have created a basic, progressively enhanced, off-canvas pattern.

Whenever you create interaction with JavaScript and modern CSS features such as transforms, it makes sense to consider things from the least powerful device you want to support. What about the two people who don't have JavaScript (yes, those guys), or if there is a problem with the JavaScript loading or executing? What if somebody prefers reduced motion? Don't worry. It's possible, with a little effort, to ensure a working interface for every eventuality.

When building these kinds of interface patterns, I find it most useful to start with the lowest set of features and enhance from there. First, establish what someone sees if they don't have JavaScript available. After all, it's no use parking a menu off-screen when JavaScript isn't available if the method for displaying the menu relies on JavaScript. Instead, in this case, we are relying on markup to place the navigation area in the normal document flow. Worst case, whatever the viewport width, they can merely scroll down the page and click a link:



A basic off-canvas menu

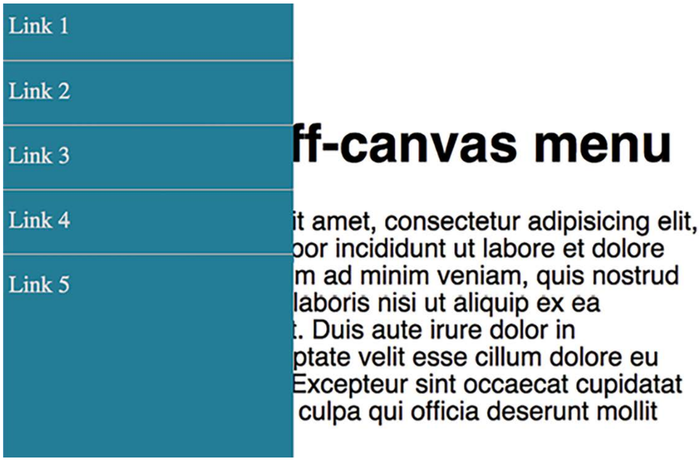
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 12.19: A basic but usable default layout

If JavaScript is available, for smaller screens, we "pull" the menu off to the left. When the menu button is clicked, we add a class to the `body` tag (with JavaScript) and use this class as a hook to move the navigation back into view with CSS `transform3d()`. This will provide the best visual performance, as it's a process typically offloaded to the graphics processor, which excels at such tasks:

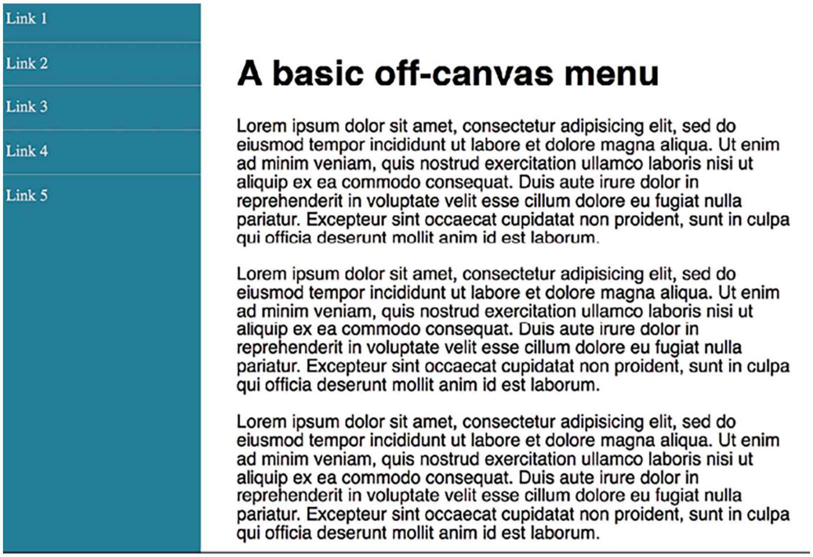


ff-canvas menu

it amet, consectetur adipisicing elit,
oor incididunt ut labore et dolore
m ad minim veniam, quis nostrud
laboris nisi ut aliquip ex ea
. Duis aute irure dolor in
ptate velit esse cillum dolore eu
Excepteur sint occaecat cupidatat
culpa qui officia deserunt mollit

Figure 12.20: With a click, we slide over our menu

For larger viewports, we hide the menu buttons and merely position the navigation to the left and move the main content over to accommodate:



A basic off-canvas menu

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 12.21: Where screen size allows, we can show the menu by default

Finally, to cater to anyone who prefers reduced motion, we use a media query to prevent movement when the menu is opened and closed:

```
@media (prefers-reduced-motion) {
  .navigation-menu {
    transition: none;
  }
}
```

If you open the example in the browser, you can simulate the different eventualities with the developer tools by changing the `js` class on the `html` element to `no-js` to see how it would look without JavaScript. Also, you can emulate `prefers-reduced-motion` in Chrome by opening the Dev Tools Command Palette and typing `prefers-reduced-motion`, and then clicking the option to emulate it.

This is a simplified example, but hopefully it's apparent that embracing a progressive enhancement approach ensures the widest possible audience will get a workable experience from your design. Remember, your users don't need visual parity, but they might appreciate functional parity.

Animating with CSS

If you've worked with applications such as Final Cut Pro or After Effects, you'll have an instant advantage when working with CSS animations. CSS employs animation keyframing conventions found in timeline-based applications.

If you have never worked with keyframes or even come across the term, here is all you need to know. When you are devising an animation, you will choose key moments where things need to be in a certain position. Imagine a bouncing ball. At first, it is in the air, which would be one **keyframe**, and then it is on the floor, another keyframe. When you specify keyframes, the animation knows how to fill in the blanks between them and create the animation.

There are two components to a CSS animation: first, writing a set of keyframes inside an `@keyframes` at-rule declaration, and then employing that keyframe animation with the `animation` property and associated values. Let's take a look.

In a previous example, we made a simple flip effect on elements that combined transforms and transitions. Let's bring together all the techniques we have learned in this chapter and add an animation to the previous example. In the following example, `example_12-08`, let's add a pulsing animation effect around our element once it has flipped.

Firstly, we will create a keyframes at-rule:

```
@keyframes pulse {
  100% {
    text-shadow: 0 0 5px white;
    box-shadow: 0 0 3px 4px #f90;
  }
}
```

As you can see, after writing `@keyframes` to define a new keyframe at-rule, we name this particular animation (`pulse`, in this instance). The `@keyframes` at-rule describes what you want to happen for each cycle of your animation loop.

It's generally best to use a name that represents what the animation does, not where you intend to use the animation, as a single `@keyframes` rule can be used as many times as you need throughout a project.

We have used a single **keyframe selector** here: `100%`. However, you can set as many keyframes (defined as percentage points) as you like within an `@keyframes` at-rule. Think of these as points along a timeline. For example, at 10%, make the background blue, at 30%, make the background purple, and at 60%, make the element semi-opaque. On and on as you need.

> There is also the keyword `from`, which is equivalent to 0%, and `to`, which is equivalent to 100%. You can use the keywords like this:
>
> ```
> @keyframes pulse {
>   to {
>     text-shadow: 0 0 5px #bbb;
>     box-shadow: 0 0 3px 4px #bbb;
>   }
> }
> ```

You'll notice here that we haven't bothered to define a starting point. That's because the starting point is the state each of those properties is already at. Here's the part of the specification that explains that (`https://www.w3.org/TR/css3-animations/#keyframes`):

"If a '0%' or 'from' keyframe is not specified, then the user agent constructs a '0%' keyframe using the computed values of the properties being animated. If a '100%' or 'to' keyframe is not specified, then the user agent constructs a '100%' keyframe using the computed values of the properties being animated."

In this `@keyframes` at-rule, we've added `text-shadow` and `box-shadow` at `100%`. We can then expect the keyframes, when applied to an element, to animate the text shadow and box shadow to the defined amount. But how long does the animation last? How do we make it repeat, reverse, and more? This is how we actually apply a keyframes animation to an element:

```
.flipper:hover .flipper-horizontal {
  transform: rotateY(180deg);
  animation: pulse 1s 1s infinite alternate both;
}
```

The `animation` property here is a shorthand for several animation-related properties. In this example, we are declaring (in order) the following:

1. The name of the keyframes declaration to use (`pulse`)
2. The `animation-duration` value (1 second)
3. The `animation-delay` value before the animation begins (1 second, to allow time for our button to first flip)
4. The number of times the animation will run (infinitely)
5. The direction of the animation (`alternate`, so it animates first one way and then back the other)
6. That we want `animation-fill-mode`, which we will cover in detail momentarily, to retain the values that are defined in the keyframes, whether going forward or backward (`both`)

The shorthand property can actually accept all seven animation properties. In addition to those used in the preceding example, it's also possible to specify `animation-play-state`. This can be set to `running` or `paused` to effectively play and pause an animation. Of course, you don't need to use the shorthand property; sometimes, it can make more sense (and help when you revisit the code in the future) to set each property separately. The following are the individual properties and example values. Where appropriate, alternate values have been listed with a comment:

```
.animation-properties {
  animation-name: warning;
  animation-duration: 1.5s;
  animation-timing-function: ease-in-out;
  animation-iteration-count: infinite;
  animation-play-state: running; /* could also be 'paused' */
  animation-delay: 0s;
  animation-fill-mode: none; /* could also be 'forwards', 'backwards' or
  'both' */
  animation-direction: normal; /* could also be set to 'reverse',
  'alternate' or 'alternate-reverse' */
}
```

You can read the full definition for each of these animation properties in the CSS Animations Level 1 specification at `https://www.w3.org/TR/css-animations-1/`.

You can run multiple animations on an element with the shorthand property by comma-separating them, as in this example:

```
animation: animOne 1s alternate both, animTwo 0.3s forwards
```

As mentioned previously, it's simple to reuse a declared `keyframes` on other elements and with completely different settings:

```
.flipper:hover .flipper-vertical {
  transform: rotateX(180deg);
  animation: pulse 2s 1s cubic-bezier(0.68, -0.55, 0.265, 1.55) 5
    alternate both;
}
```

Here, the `pulse` animation would run over 2 seconds and uses an `ease-in-out-back` timing function (defined as a `cubic-bezier` curve). It runs five times in both directions. This declaration has been applied to the vertically flipping element in the `example_12-08/styles.css` file.

The animation-fill-mode property

The `animation-fill-mode` property is worthy of a special mention. Consider an animation that starts with a yellow background and animates to a red background over 3 seconds. You can view this in `example_12-09`.

We apply the animation like this:

```
.background-change {
  animation: fillBg 3s;
  height: 200px;
  width: 400px;
  border: 1px solid #ccc;
}
@keyframes fillBg {
  0% {
    background-color: yellow;
  }
  100% {
    background-color: red;
  }
}
```

However, once the animation completes, the background of the `div` will return to nothing. That's because, by default, when an animation ends, the element goes back to exactly how it was before the animation ran. In order to override this behavior, we have the `animation-fill-mode` property. In this instance, we could apply this:

```
animation-fill-mode: forwards;
```

This makes the item retain any values that have been applied by the animation end. In our case, the `div` would retain the red background color that the animation ended on. More on the `animation-fill-mode` property can be found here: `https://www.w3.org/TR/css-animations-1/#animation-fill-mode`.

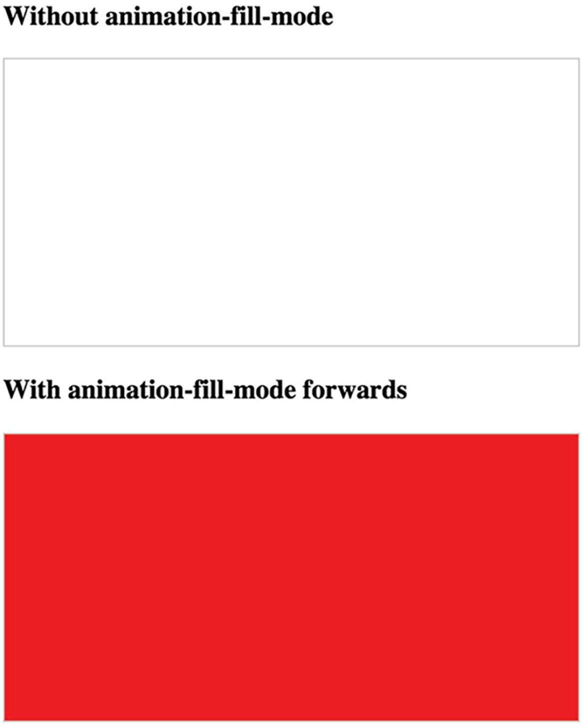

Without animation-fill-mode
With animation-fill-mode forwards

Figure 12.22: Without animation-fill-mode set to forward, animations run and then immediately reset

Using @starting-style to transition items entering the DOM

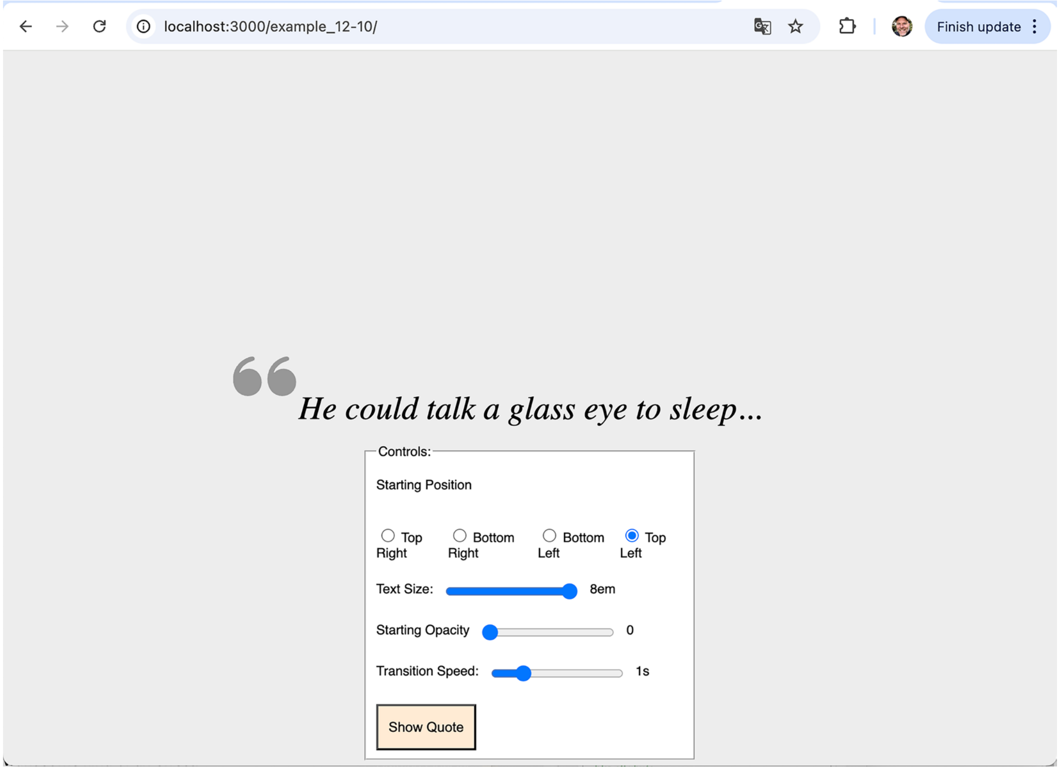
Remember, near the beginning of the chapter, I mentioned that newer browsers support a means for transitioning items into the DOM? If you do remember, great, because this is where I tell you how. If you don't remember, don't worry, I'm still going to tell you.

As we have already discussed, ordinarily, items entering the DOM (typically by JavaScript) appear instantly. But the reality is, often when we are building applications, when we add something to the DOM, we want some kind of a visual effect to transition them in. Imagine a list item being added to a to-do application. It would be nice if the new item grows into the space, instead of merely snapping. That's the kind of thing `@starting-style` exists for.

The `@starting-style` at-rule lets you wrap some styles for an element, and these are the styles that the element is set to the moment it enters the DOM. With a transition applied (`@starting-style` is a transition-related at-rule, not animation-related), the normal styles for the element then take over, and the new element transitions to its default position—the styles outside of `@starting-style`.

Be aware that you need to ensure the starting styles are written after the default rules in the style sheet; otherwise, the `@starting-style` rules get overwritten by the default ones. This is particularly important if you are adding a class to show your element, where this invariably adds more specificity to a display setting. In this situation, you likely need to add `@starting-style` on the compound selector to ensure its specificity is equal. That is how it has been applied in `example_12-12`.

Let's look at an example. You can play with this by opening `example_12-10` in the example code.



localhost:3000/example_12-10/

He could talk a glass eye to sleep…

Controls:

Starting Position

○ Top Right ○ Bottom Right ○ Bottom Left ● Top Left

Text Size: 8em

Starting Opacity 0

Transition Speed: 1s

Figure 12.23: @starting-style provides a simple means of gracefully bringing elements into view

The example is essentially a page and a button. Click the **Show Quote** button, and we add an element to the page with a little text inside. Each time the button is clicked, the item is removed from the DOM and re-added.

This example is contrived; when adding something like this to the DOM, we might well be better off implementing it with something like the `dialog` element, which we covered at the end of *Chapter 2*, *Writing HTML Markup*. But to keep things simple for the example, we are just using `div`s.

The controls at the bottom of the example merely allow you to tweak the style of the element as it enters the DOM. They amend the values assigned to custom properties, which are in turn referenced in the CSS. Once you click to see the quote, you can see the amended styles as it enters. In the style sheet, the `@starting-style` section is nested inside the styles for the element that shows the text:

```
@starting-style {
  transform: translate(var(--pos));
  font-size: var(--size);
  opacity: var(--opacity);
  left: auto;
  top: auto;
}
```

You can wrap a number of elements in `@starting-style` to set a number of elements' starting styles in one go, in the same manner you wrap things with any other at-rule we have looked at. Or, and as I tend to use it, you can nest the starting styles for a particular selector inside its rule, just as you might nest a related media query.

But you can't animate them being removed

Sadly, there is no simple way of animating elements as they are removed from the DOM. For this situation, it is necessary to set or amend a class or attribute on an element, run an exit animation/transition, and then remove it from the DOM once the animation ends, by either listening for the `animationend` event or using a `setTimeout` function in JavaScript.

Transition elements to display: none with transition-behavior

If the element you want to show and hide is remaining in the DOM, you can use `transition-behavior: allow-discrete` to easily transition it off the page using `visibility: hidden` or `display: none`.

As the `transition-behavior` property name implies, this is a property that is used on transitions, and not animations.

Here is an example of using `@starting-style` and `transition-behavior: allow-discrete`, to show an element gracefully appearing and disappearing from view. You can take a look at the chapter code `example_12-12`.

Here is the markup:

```
<button>Toggle Boxes</button>
<div class="box without">Without</div>
<div class="box with">With</div>
```

And here is the CSS:

```
.box {
  display: none;
  opacity: 0;
  height: 200px;
  aspect-ratio: 1;
  background-color: red;
  transition: all 3s;
  color: white;
  font-family: sans-serif;
}

.without {
  &.show {
    display: inline-block;
    opacity: 1;
  }
}

.with {
    transition-behavior: allow-discrete;
    /*  Note if you don't include starting-style nested inside the most
specific related selector, it will be overridden due to low specificity */
  &.show {
    display: inline-block;
    opacity: 1;
    @starting-style {
      opacity: 0;
    }
  }
}
```

And then here is a dab of JavaScript to toggle the class on the boxes:

```
const btn = document.querySelector("button");
const boxes = [...document.querySelectorAll(".box")];
btn.addEventListener("click", () => {
    boxes.map((box) => {
        box.classList.toggle("show");
    });
});
```

We have a button and two boxes, both set to `display: none` by default. The button is attached to a little JavaScript to toggle a class onto both our hidden boxes. The first one does not have `@starting-style` or `transition-behavior: allow-discrete` applied; the second does. Notice how the first snaps in and out of view, while the second one transitions gracefully in and out? I've set an extended period of time on the transition in the code to illustrate, but a shorter time of 0.25s–0.35s creates a very elegant way to bring elements in and out of the interface. It's a great addition to CSS.

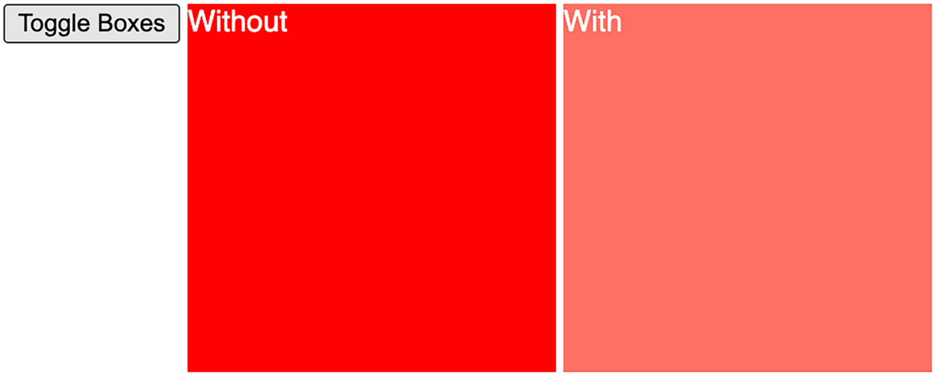


Figure 12.24: With @starting-style and allow-discrete, we have a solid mechanism for elegantly showing and hiding elements on a web page

Animating height with the calc-size() function

For as long as I can remember, one thing that has infuriated developers is the lack of a simple means of animating an element to its native height. It's a common requirement where you want to animate the opening and closing of a drawer with variable content. Finally, there looks to be a simple solution on the horizon that does not require JavaScript. The only downside is that, as I write this, it is only supported in Chrome and Edge. As it is still experimental, and the syntax is likely to change, we won't cover it in great depth here, but the short version is that it should soon be able to animate the opening/closing of something simply like this:

```
div {
  transition: .3s;
  height: 40px;
  overflow: clip;
}

div:hover {
  height: calc-size(auto, size);
}
```

With the `calc-size` function, `auto` is the intrinsic size (you might use `min-content`, `max-content`, or `fit-content` instead), and then `size` is the result of that, which you then might choose to do additional calculations with. It's looking like a promising addition to our CSS toolkit, in that we will likely get the functionality at long last, even if the particulars of the syntax are tweaked before it is supported across all browsers.

The animation-composition property

Unless you are regularly tasked with authoring complex animations, the `animation-composition` property is not something you likely need to concern yourself with. The `animation-composition` property allows you to define what kind of composition technique to use when an element has multiple animations running on it at the same time. There are three values to consider when using the property: `accumulate`, `add`, and `replace`, with the latter being the default value. Here is what the other two do to multiple animations that are applied to an element at the same time:

- `accumulate` effectively gathers the similar properties that already exist, and combines the values set on them
- `add`, on the other hand, effectively takes what exists on the element and applies the next animation as another step to it

This property is not something I find myself needing often, but as is often the case with CSS, knowing something exists for this reason might dig you out of a future UI design hole. You can see the difference these different values apply in `example_12-11`.

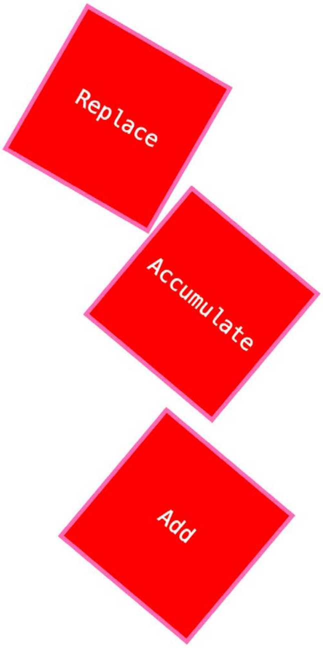


Figure 12.25: The animation-composition properties create subtly different results

Summary

It would be entirely possible to fill multiple books covering the possibilities of CSS transforms, transitions, and animations. Hopefully, by dipping your toe in the water with this chapter, you've been able to pick up the basics and run with them.

In this chapter, we've learned what CSS transitions are and how to write them. We got a handle on timing functions such as `ease` and `linear`, and then used them to create simple but fun effects.

We then learned all about 2D transforms, such as `scale` and `skew`, and how to use them in tandem with transitions. We also looked briefly at 3D transformations before learning all about the power and relative simplicity of CSS animations. You'd better believe our CSS muscles are growing!

One topic related to animations we didn't look at in this chapter is `@property`, which allows you to define your own CSS custom property, something that can be handy when animating elements. But we will save that for the next chapter, where we are going to be looking in depth at CSS custom properties and functions. These are more recent additions to CSS, in comparison to things such as media queries, so we will see how, in many responsive situations, newer functions such as `clamp()` provide even better responsive solutions than media queries.

Exercises and training

At this point, you might find it useful to find a site using transitions, transforms, and animations and try playing with the settings in the developer tools. Inspect any moving elements and then tweak the relevant values and properties. Can you send elements in the opposite direction? Can you make transitions take longer or shorter? Are there any 2D transforms you can amend to use the `transform3d()` function?

You can make a start on `https://rwd.education`—I won't mind!

13

Custom Properties and CSS Functions

Some years ago, I wrote a book about Sass, a superset language for CSS. One of the main selling points of Sass, alongside things such as nesting, which we looked at in *Chapter 6*, *CSS Nesting, Layers, Selectors, and More*, was that it allowed you to store often-used values as **variables**. Things such as colors could be named more simply as words rather than hex or RGB values, and that allowed for far easier reuse of those values.

Some years later, CSS got custom properties, which at first glance seemed like nothing more than a way to have variables in CSS, without needing a superset language such as Sass. It turns out that CSS custom properties are far more powerful than Sass variables. So powerful, in fact, that this whole chapter is largely dedicated to them.

Another, newer CSS addition that goes hand in hand with custom properties is the ability to manipulate values, including custom properties, with mathematical functions. Exploring these functions will also form a substantial part of this chapter.

As we will see, alongside viewport-related length values such as `vh` and `vw`, these capabilities enable us to solve a multitude of responsive problems that would otherwise remain, well, problems!

We will begin by understanding custom properties, and then learn how to manipulate them and other values with CSS functions.

Here are the main areas we will cover in this chapter:

- What a custom property is
- Special `env()` custom properties
- Setting a custom property with CSS and JavaScript
- Using a custom property with CSS and JavaScript
- Understanding CSS functions
- Reducing the need for media queries
- `calc()`, `clamp()`, `min()`, and `max()`
- Examples of using custom properties with CSS functions

Working with custom properties

Let's start with a simple use case, storing a `font-family` name that we can then reference more simply later in the style sheet. This is what a custom property definition looks like:

```
:root {
  --MainFont: "Helvetica Neue", Helvetica, Arial, sans-serif;
}
```

Here, we are using the `:root` pseudo-class to store the custom property in the document root. You don't have to store them in the root, but storing them here means that any descendant element has access to them.

But custom properties respect the cascade. As we will see shortly, being able to reassign custom properties in different scopes is one of their superpowers.

The `:root` pseudo-class always references the topmost parent element in a document structure. In an HTML document, this would always be the HTML tag, but for an SVG document (we will look at SVG in *Chapter 11*, *SVG*), it would reference a different element.

When we set a custom property, the syntax is that it should always begin with two dashes, then the custom name, like this: `--MainFont`. After the end of the property, just like a normal CSS property, we terminate the property with a colon. Then, we just assign whatever value we want to it in the normal manner.

It is case-sensitive, so `--MainFont` would be a different custom property from `--mainFont`.

We can reference that value elsewhere in our style sheet with the `var()` notation, like so:

```
.Title {
  font-family: var(--MainFont);
}
```

And the browser would effectively read that as an instruction to do this:

```
.Title {
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
}
```

You can store as many custom properties as you need in this manner.

Even though, in this example, we are saving some characters using the custom property, making use of them isn't merely an exercise in convenience when authoring; it's an approach that pays you back time and again when you want to change things.

Imagine that you have defined and referenced colors and font declarations throughout your project in this manner, and then some time later, you want to change a color or `font-family`. It's now just a single change in one place, and the values effectively get changed everywhere. Every rule that makes use of the variable gets the new value, without having to amend the rules directly.

Simple theming

The ability to set a value in one place to be used in many other places makes custom properties a great candidate for theming. And we are not limited to CSS as a means of interacting with them, as there is a JavaScript API to get and set the values of custom properties.

Let's look at a simple example of using and switching custom properties with JavaScript. You can find this in `example_13-01`. We will make a page with the poem *If*, by Rudyard Kipling. At the bottom is a simple light/dark mode toggle button. All the button will do is toggle the value of two CSS custom properties, `--background` and `--foreground`.

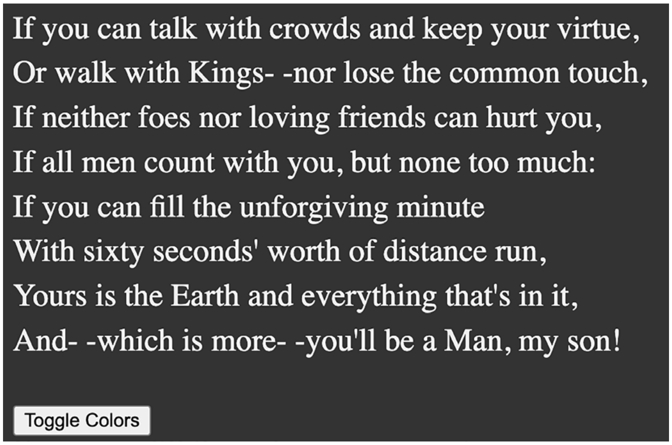
Here is our CSS:

```
body {
  background-color: var(--background);
  color: var(--foreground);
}
```

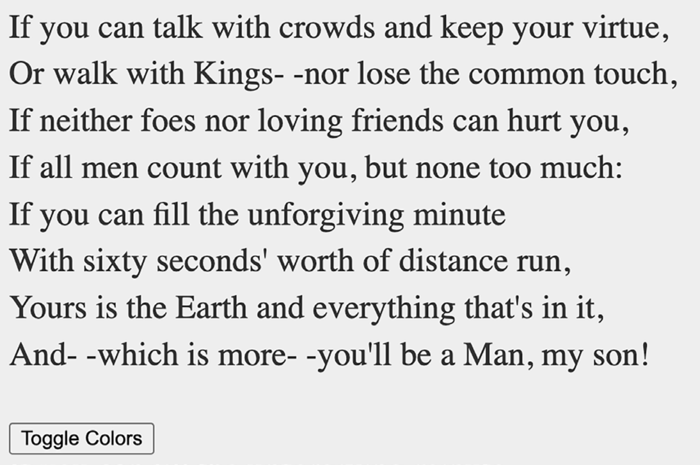
The following is the snippet of JavaScript we are using. Essentially, it just says that when someone clicks the toggle button, if the `foreground` variable is currently #eee (nearly white), then set it to #333 (a dark gray); otherwise, set it to #eee. And, if the `background` variable is #333, set it to #eee; otherwise, set it to #333:

```
var root = document.documentElement;
var btn = document.getElementById("colorToggle");
btn.addEventListener("click", (e) => {
    root.style.setProperty(
        "--background",
        getComputedStyle(root).getPropertyValue("--background") === "#333"
            ? "#eee"
            : "#333"
    );
    root.style.setProperty(
        "--foreground",
        getComputedStyle(root).getPropertyValue("--foreground") === "#eee"
            ? "#333"
            : "#eee"
    );
});
```

Here is a screenshot showing each state:



If you can talk with crowds and keep your virtue,
Or walk with Kings- -nor lose the common touch,
If neither foes nor loving friends can hurt you,
If all men count with you, but none too much:
If you can fill the unforgiving minute
With sixty seconds' worth of distance run,
Yours is the Earth and everything that's in it,
And- -which is more- -you'll be a Man, my son!



If you can talk with crowds and keep your virtue,
Or walk with Kings- -nor lose the common touch,
If neither foes nor loving friends can hurt you,
If all men count with you, but none too much:
If you can fill the unforgiving minute
With sixty seconds' worth of distance run,
Yours is the Earth and everything that's in it,
And- -which is more- -you'll be a Man, my son!

Figure 13.1: An example of each state side by side, colors easily swapped with custom properties

Setting and getting custom property values with JavaScript

If you only take two things from that prior JavaScript snippet, it should be the manner in which you can both **get** and **set** custom properties. At first glance, it looks like you should be able to set a custom property to any string value you like. But it is a little stricter than that.

Suppose we have an element with a class of `heading-information`. We could try and set a custom property on that element with JavaScript like this:

```
const headerInfo = document.querySelector(".heading-information");

headerInfo.style.setProperty("--doILikeScones", "You bet I do!");

console.log(getComputedStyle(headerInfo).getPropertyValue(
    "--doILikeScones")); // hoping it will log out "You bet I do!"
```

But that won't work. Why? Well, custom properties must have their value set to a valid CSS value, so you can't use them to pass around any arbitrary snippet of data. If we amended our code and set the value to, say, `"100px"`, we would then get that value logged out correctly.

So, that's the only real rule to be aware of. You *set* and *get* values in JavaScript as strings, but in order for those strings to *work*, they must be valid CSS values.

Then, assuming you have set the property to something valid, we *get* a value, using the `getPropertyValue()` method like this:

```
const headerInfo = document.querySelector(".heading-information");
const doILikeSconesValue =
    getComputedStyle(headerInfo).getPropertyValue("--doILikeScones");
console.log(doILikeSconesValue); // would log out value of custom property
```

And everything will work as expected.

Custom properties and specificity

Custom properties also behave like other properties in terms of the cascade and specificity. And that can be incredibly useful. For example, we set our custom properties at the root level, but if they are redeclared closer to the element in question, the more specific value will override the first, as they inherit values from the closest ancestor. Consider this CSS:

```
:root {
  --backgroundColor: red;
}

header {
  --backgroundColor: goldenrod;
}
```

The `header` and any elements within it that make use of the `--backgroundColor` custom property will have a “goldenrod” background color, whereas other elements will have a red background.

In our example, we have just set a background color, but it is possible to set all manner of defaults in one place for items to inherit, and then override as necessary.

You could do something like this:

```
:root {
  --fontSize: 1.2em;
  --lineHeight: 1.4;
  --col1: #333;
  --col2: #f9f9f9;
  --fontFam: -apple-system, BlinkMacSystemFont, Helvetica Neue, Segoe UI,
      Tahoma, Roboto, Oxygen, Ubuntu, Cantarell, Fira Sans, Open Sans,
        sans-serif;
```

```
    --padInline: clamp(10px, 3vw, 25px);
    --padBlock: clamp(10px, 2vh, 20px);
  }
```

And then anywhere you want a visual component that uses those properties, you can pick and choose them:

```
  .component {
    font-size: var(--fontSize);
    font-family: var(--fontFam);
    line-height: var(--lineHeight);
    color: var(--col1);
    background-color: var(--col2, #000);
    --padInline: 15px;
  }
```

Remember that you can also reassign them for any child elements.

Some things to note from those code blocks. You may have noticed `clamp()` in there. We haven't looked at that yet, but we will later in the chapter. We have also done something a little different with `background-color`, and that's setting a fallback value. Let's talk about that a little more next.

Setting a fallback value

There may be a situation where you want to protect against a custom property being unavailable. Perhaps the values are set with JavaScript, and that has failed to load, or hasn't been initialized yet. Whatever the reason, you can safeguard against the eventuality by providing a fallback value. The syntax is straightforward: simply provide your fallback value after the custom property name and a comma.

For example, suppose I wanted to use the `--backgroundColor` custom property, but default to a dark gray color if that variable was unavailable. I could write it like this:

```
  .my-Item {
    background-color: var(--backgroundColor, #555);
  }
```

If you are ever unsure whether a custom property will be set, it makes a lot of sense to get into the habit of supplying fallback values whenever you use custom properties. That way, if the visual component you are building finds itself in an area where the custom property is not set, it will still render with the sensible default you provided.

env() environment variables

In addition to custom properties we might make for ourselves, there are also some system-level properties that can be read in from the environment we are operating in. These are called **environment variables.** The idea here is that there are certain values that only the host environment knows. This gives us the ability to factor in that unknown when we build, so regardless of the environment, our responsive designs can respond.

The only solid example of these I'm currently aware of is the `safe-area-inset-*` properties applicable to notched mobile phones. Made famous by the iPhone X, the "safe-areas" relate to the sections at the top and bottom of the screen where areas of UI or physical hardware, such as cameras, impinge upon the viewing area.

Environment variables are defined with the `env()` function rather than the `var()` function for custom properties. Values are passed into this function just like you would any other CSS function. For example, if we wanted to add padding to the top of our element equal to the height of the environment's "safe area inset top," we could do this:

```css
padding-top: env(safe-area-inset-top);
```

If the browser has that value available and it understands the `env()` function, it will apply the value; otherwise, it will skip over it to the next declaration.

Unlike custom properties, we are not able to set environment variables for ourselves; these are values set by the environment alone.

You can read the current specification for environment variables here: `https://drafts.csswg.org/css-env-1/#env-function`.

Custom properties and web components

Web components provide the web platform with native encapsulated components. This means the styles within them are contained, unable to access styles above them in the DOM. However, one thing that can penetrate this boundary is – you guessed it – custom properties.

This makes custom properties the perfect mechanism for passing around values that you want to share among nested components.

You can set a custom property on the outermost part of a web component using the `:host` pseudo-class, like this:

```
:host {
  --myCustomProperty: 100%;
}
```

And then that `host` component, and any nested components, can access that custom property.

> As interesting and useful as they are, we won't delve further into web components in this book, as they are a little out of scope. For a good primer on web components, I'd recommend taking a look at `https://developer.mozilla.org/en-US/docs/Web/Web_Components`.

So, now we are at a point where we know about custom properties, what they look like, and how we can set them at different places and use specificity to override them. We have also looked at setting defaults, just in case a custom property isn't set elsewhere. We have also looked at environment variables and where they may be a consideration.

It's time now to start manipulating these values with functions. Manipulating values provides a whole slew of additional capabilities and flexibility. Let's take a look.

CSS functions

There are lots of CSS functions; we've looked at many already in prior chapters. Remember, anything in CSS that is a word followed by parentheses, such as `rotate()` or `drop-shadow()`, is a CSS function. And there are far more CSS functions than I have ever used. There are functions for counters, colors, grids, shapes, and even more esoteric math functions such as `pow()`, `mod()`, and `round()`. However, the ones we are interested in for this chapter are the ones I have found particularly useful in solving real responsive web design problems.

The fact is, many of the CSS functions that have come along in recent years better media queries in some situations.

Want to have text that is no smaller than 16px, but then scales with the size of the viewport, yet never gets bigger than 30px?

With media queries, you would have to try to solve that problem with something like this:

```
.headline {
    font-size: 16px;

    @media (min-width: 400px) {
        font-size: 6vw;
    }

    @media (min-width: 1000px) {
        font-size: 30px;
    }
}
```

But the reality is that's quite brittle. You'll find yourself adding lots of little "tweak points" where you need to add another media query. For example, when the viewport is 950px wide, that 6vw looks a little comically big.

Instead, we can use the `clamp()` CSS function and do this:

```
.headline {
  font-size: clamp(16px, 4vw, 30px);
}
```

That's pretty powerful, right? The `clamp()` function is just one of many CSS functions we now have at our disposal. But hang on. I got a little excited then and jumped ahead. Before we look in detail at functions such as `clamp()`, let's start by looking at a function you may have already looked at, `calc()`.

calc()

The `calc()` function allows us to do basic arithmetic in CSS. I purposely said *basic* because it only allows addition, subtraction, division, and multiplication.

It's essential to put whitespace around the plus and minus symbols; otherwise, the `calc()` won't, er… calc! It isn't necessary to have whitespace around the division or multiplication symbols, but I always add it there too, just for consistency.

So, to illustrate, this is good:

```
.thing {
  width: calc(100vw - 30px);
}
```

But this is bad:

```
.thing {
  width: calc(100vw -30px);
}
```

As you can see in the preceding example, you can mix and match units in the calculations. This is particularly useful in our responsive challenges. Suppose we are building some piece of user interface and we want something the full height of the dynamic viewport, but minus a fixed height header that is 50px. We can solve this with ease with `calc`:

```
.thing {
  height: calc(100dvh - 50px);
}
```

Actually, that is not so dissimilar to the first example. So, how about we up the ante and factor in environment variables? That would look like this:

```
.thing {
  height: calc(
    100vh - env(safe-area-inset-top) - env(safe-area-inset-bottom) - 50px
  );
}
```

And now our UI is the full height of the viewport (`100vh`), minus any safe area the device has set, minus the height of our fixed height header. Now that is a responsive piece of interface!

And as you might imagine, you can use custom properties inside `calc()` too.

> There are plenty of other mathematical expressions in the latest *CSS Values* specification, including trigonometry and exponential functions: `https://drafts.csswg.org/css-values/#trig-funcs`.

Now, as we go ahead and look at some of the "comparison" functions of CSS, notice that they have the `calc()` capability baked in. You don't need to use `calc()` in them explicitly; you can just use the basic math operations (addition, subtraction, multiplication, and division) in there directly. We will make use of this capability as we look at each function in turn.

First up, `min()`.

min()

The `min()` function accepts any number of comma-separated arguments and resolves to the minimum.

Consider this:

```
.my-responsive-component {
  width: min(100vw, 700px);
}
```

Let's suppose we have our browser window `800px` wide. Looking at the preceding code, can you guess how wide our element will be? In that scenario, it will be `700px` because 100vw would be 800px. However, if we reduced our browser window width to, say, `600px`, it would resolve to `600px` (`100vw`) as that would now be the minimum value of the two.

Remember, we also have basic math functionality in `min()`, so if we wanted to make allowance for some responsive part, we can do that too. Here is the example amended to use a combination of hard and dynamic values:

```
.my-responsive-component {
  width: min(100vw, 700px - 5%);
}
```

And remember, we are not limited to just two arguments:

```
.thing {
  width: min(50vw, 400px, 100% - 10px);
}
```

max()

Having just looked at `min()`, I'm sure you have a fair idea of how `max()` works. It takes any number of arguments and resolves to the largest.

Much like `clamp()`, which we will look at in more detail next, I find it most useful when I want text to smoothly scale in a responsive scenario but still have a minimum size for accessibility. For example, if I never want the text in a label to be less than 18px, I use this:

```
label {
  font-size: max(4vw, 18px);
}
```

It can be confusing using these functions at first, as you are often using the math function that seems to have a name that is the opposite of what you are using it for! Here we are using `max()` to set a minimum size for something.

If the same happens for you, you'll just have to trust me that it gets less confusing the more you use them.

If it's still making your brain hurt, just make sure that whatever value you want as a minimum is chucked inside a `max()` function, as the value is never going to resolve to less than that.

clamp()

Besides `calc()`, the CSS function I tend to reach for the most in responsive work is `clamp()`.

It came to browsers slightly later than `min()` or `max()` but made up for its tardy arrival by solving a classic design conundrum. You know, the "I want the headline text to be flexible in size, but it can't go too small. And it can't go too big either." Yes, that "Goldilocks" one!

The `clamp()` function lets us specify, in the order we pass them into the function, a minimum, variable, and maximum size for any length. For example, we can address our Goldilocks text sizing problem with `clamp` like this:

```
.hero-text {
  font-size: clamp(32px, 40vw, 80px);
}
```

And while our headline text might vary in size, depending on the viewport, it will never be less than 32px or larger than 80px. Beautiful!

Like all the mathematical functions we have looked at, you can use custom properties in there, too. Something like this would allow you to have a site-wide minimum and maximum text size set with a custom property, with any kind of responsive-friendly unit in the middle:

```
.hero-text {
  font-size: clamp(var(--textMin), 40vw, var(--textMax));
}
```

And we can nest some calculations in there, too, if needed:

```
.hero-text {
  font-size: clamp(var(--textMin), 40vw + 5rem, var(--textMax));
}
```

Keep in mind that `clamp()` isn't just for text; you can use it in any place that accepts a length value.

You can read the specification for `clamp()` here: `https://www.w3.org/TR/css-values-4/#funcdef-clamp`.

Putting it all together

Let's make a fun example, bringing together some of the things we have looked at in previous chapters.

With just a couple of elements, let's simulate the basics of a basketball court, and the ball bouncing and being passed around.

Here is the markup, probably the least important part of this example:

```
<div class="court">
  <div class="ball"></div>
</div>
```

Let's make a vague court shape and give it a background color, something like a basketball court:

```
.court {
  width: 400px;
  height: 200px;
  background-color: rgb(197, 150, 86);
  position: relative;
}
```

Now, we want to add a ball to our court. This ball will simulate a bounce with an animation and transition its position, which we will set with custom properties. Now, if for some reason you haven't read *Chapter 12, Transitions, Transformations, and Animations*, yet (shame on you), all you need to know is that this animation makes a circle appear as if bouncing by making it scale up in size and then back down:

```
.ball {
  --ballSize: 18px;
  --ball-border: #fff;
  --ball-color: #df4a23;
  position: absolute;
  left: calc(var(--x) - var(--ballSize));
  top: calc(var(--y) - var(--ballSize));
  width: var(--ballSize);
  height: var(--ballSize);
  background-color: var(--ball-color);
  border-radius: 50%;
  transition: all 0.3s ease-in-out;
  border: 1px solid var(--ball-border);
  animation: 0.5s ease infinite alternate forwards bounce;
}

@keyframes bounce {
  0% {
    transform: scale(1);
  }
  100% {
    transform: scale(1.3);
  }
}
```

You can see that we are setting some custom properties on the ball. Particularly useful is `--ballSize`. We reference that a few times, which means that tweaking that value automatically changes both the width and height of the ball, and also factors into the calculations being made to the left and top positions. In those `calc()` functions, you can also see there is a custom property for `--x` and `--y`, which we are setting every so often with JavaScript. Here is the snippet of JavaScript that powers our ball movement:

```
// Get the ball from the DOM
const ball = document.querySelector(".ball");
// Get a random number between the Min and the Max
const getRand = (min, max) => Math.floor(Math.random() * (max - min + 1) +
min);

// Set the values into the custom property
const setCoords = () => {
    ball.style.setProperty("--x", `${getRand(0, 100)}%`);
    ball.style.setProperty("--y", `${getRand(0, 100)}%`);
};

setCoords();
// Every so often, set the position of the ball
```

```
setInterval(() => {
    setCoords();
}, getRand(1000, 3000));
```

If you look at the DOM in `example_13-02`, you can see that the value is updated every 1 to 3 seconds. Because we have a transition set on `all`, when the ball gets its left and top position updated, we see the ball move to its new position.

Now, that's where I have parked the example. What about you? Perhaps you could take it further and use `background-image` to set and position a basketball court? Or use an image of a basketball for the ball?

If you are confident in adding more JavaScript and markup, perhaps add some sliders to interactively resize the court with custom properties. Or perhaps a color picker to change the court or ball. Or perhaps tweak the animation to use custom properties based on screen width to alter the scale of the ball. Be creative; see what fun you can have!

Summary

We've had a good look at custom properties in this chapter, and considered ways in which they can be set and read. We have learned how custom properties respect the cascade, inheriting values from their parents, a feature that can be leveraged to our advantage. We have also considered the fact that if we find ourselves working with web components, custom properties provide a means of crossing the usually impenetrable style boundary inside a component. We also looked at environment variables, a specific kind of custom property that lets us deal with the (clears throat) "practicalities" of notches and other peculiarities of specific hardware.

In the second half of the chapter, we also looked at some of the most appropriate CSS mathematical functions – from ones you have perhaps looked at before, such as `calc()`, to real time-saving functions such as `clamp()`, a function that is tailor-made to solve responsive sizing challenges.

We've spent a bit of time together at this point, so I feel safe enough to confess something. It's about forms. If there's one area of site design that I always avoid where possible, it's making forms. I don't know why; I've just always found making them a tedious and largely frustrating task. Imagine my joy when I learned that HTML5 and CSS can make the whole form building, styling, and even validating (yes, validating!) process easier than ever before. I was quite joyous. As joyous as you can be about building web forms, that is. In the next chapter, I'd like to share this knowledge with you.

14

Forms

Before HTML5, adding pieces of a user interface, such as date pickers, placeholder text, and range sliders, into forms had always needed JavaScript. Similarly, there had been no easy way to signpost to users the kind of data we expect them to input into certain fields—for example, whether we expect telephone numbers, email addresses, or URLs.

The good news is that HTML5 largely solves these common problems.

We have a few main aims in this chapter—firstly, to understand the possibilities of HTML5 form features; secondly, to understand how we can lay out forms most simply and with the latest CSS layout methods; and finally, to understand which of the HTML form elements can be easily restyled, and which should be left alone.

In this chapter, we will learn how to do the following:

- Easily add placeholder text into relevant form input fields
- Disable autocompletion of form fields where necessary
- Set certain fields to be required before submission
- Use a regular expression to define an allowed form value
- Change the color of the input cursor/caret
- Specify different input types such as email, telephone number, and URL
- Create number range sliders for easy value selection
- Place date and color pickers into a form
- Consider more form-specific selectors such as `:in-range`, `:valid`, and `:focus-within`
- Style forms using Flexbox and indicate required fields
- Make use of new, fully stylable select menus

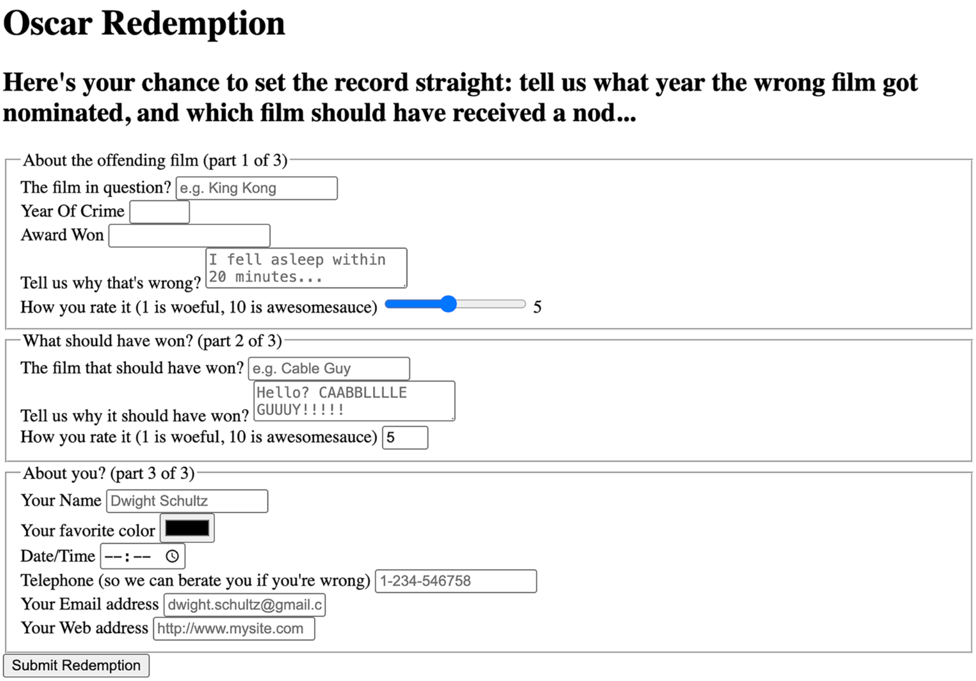
HTML5 forms

I think the easiest way to get to grips with HTML5 forms is to work our way through an admittedly contrived example form. Just like in daytime TV cooking shows, I have one I made earlier! A minor introduction is needed. Two facts: firstly, I love films. Secondly, I'm very opinionated on what is a good film and what is not.

Every year, when the film Oscar nominations are announced, I can't help but feel the wrong films have got "the nod" from the Academy of Motion Picture Arts and Sciences. Therefore, we will start with an HTML5 form that enables fellow cinephiles to vent their frustrations at the continual travesties of the Oscar nominations.

It's made up of a few `fieldset` elements, within which we are including a raft of HTML5 form input types and attributes. Besides standard form input fields and text areas, we have a number spinner, a range slider, and placeholder text for many of the fields. If you don't know what any of those elements do or look like, fear not. You will!

The HTML for this form can be found in the `example_14-01` code. Here's how it looks, with no styles applied, in Chrome:



Oscar Redemption

Here's your chance to set the record straight: tell us what year the wrong film got nominated, and which film should have received a nod...

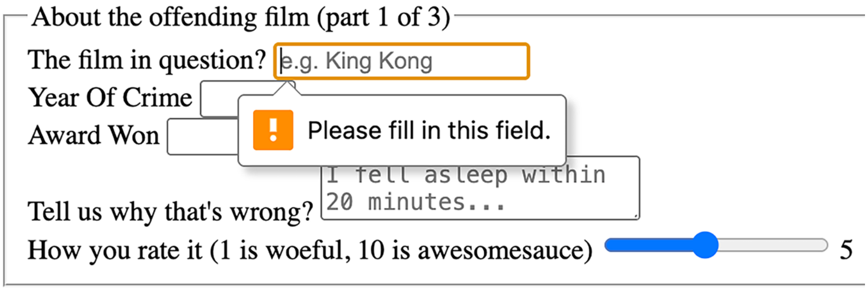
About the offending film (part 1 of 3)
The film in question? e.g. King Kong
Year Of Crime
Award Won
Tell us why that's wrong? I fell asleep within 20 minutes...
How you rate it (1 is woeful, 10 is awesomesauce) 5

What should have won? (part 2 of 3)
The film that should have won? e.g. Cable Guy
Tell us why it should have won? Hello? CAABBLLLLE GUUUY!!!!!
How you rate it (1 is woeful, 10 is awesomesauce) 5

About you? (part 3 of 3)
Your Name Dwight Schultz
Your favorite color
Date/Time --:--
Telephone (so we can berate you if you're wrong) 1-234-546758
Your Email address dwight.schultz@gmail.c
Your Web address http://www.mysite.com

Figure 14.1: A basic form with no styling

If we "focus" on the first field and start inputting text, the placeholder text is removed. If we blur focus without entering anything (by clicking outside of the input box again), the placeholder text reappears. If we submit the form (without entering anything), the following happens:



About the offending film (part 1 of 3)

The film in question? e.g. King Kong

Year Of Crime

Award Won

Please fill in this field.

Tell us why that's wrong? I fell asleep within 20 minutes...

How you rate it (1 is woeful, 10 is awesomesauce) 5

Figure 14.2: A standard browser warning from a required field

The great news is that all these user interface elements, including the aforementioned slider, placeholder text, spinner, and input validation, are being handled natively by the browser via HTML5, and no JavaScript is required.

Let's begin by getting a handle on all the capabilities of HTML5 that relate to forms and make all this possible. Once we understand all the mechanics, we can get to work styling it up.

Understanding the component parts of HTML5 forms

There's a lot going on in our HTML5-powered form, so let's break it down. The three sections of the form are each wrapped in a `fieldset`, which semantically groups the related sets of form fields, and a `legend`, which provides the textual explanation of what that `fieldset` is:

```html
<fieldset>
  <legend>About the offending film (part 1 of 3)</legend>
  <div>
    <label for="film">The film in question?</label>
    <input
      id="film"
      name="film"
      type="text"
      placeholder="e.g. King Kong"
      required
    />
  </div>
</fieldset>
```

You can see from the previous code snippet that each `input` element of the form is also wrapped in a `div` with a `label` associated with each `input` (we could have wrapped the `input` with the `label` element if we'd wanted to, too). So far, so normal. However, within this first `input`, we've just stumbled upon our first HTML5 form feature. After the common attributes of `id`, `name`, and `type`, we have `placeholder`. There are a few attributes that can be added to `input` elements that have special powers. Let's consider a few.

The placeholder attribute

As the name suggests, the `placeholder` attribute offers a means of providing a hint or some placeholder data to indicate the kind of input we are expecting. An obvious example would be the word **Search** in a search box.

To add placeholder text for an `input` element (or `textarea` element), simply add the `placeholder` attribute. Its value will be displayed in the input until the field gains focus. When it loses focus, if a value has not been entered, it will redisplay the placeholder text.

In our example, the `placeholder` attribute is filled in like this:

```
placeholder="e.g. King Kong"
```

Styling the placeholder text

You can style the text in the `placeholder` attribute with the `:placeholder` pseudo-selector:

```
input:placeholder {
  color: #333;
}
```

You can also change the `font-size` value of the placeholder text; it doesn't need to be the same as the input values that a user will enter. As ever, be mindful of accessibility. Although it is only placeholder text, I still try to ensure the text is 16px or greater.

You can also select any inputs that are currently displaying placeholder content with the `:placeholder-shown` pseudo-class. This gives you the ability to highlight fields in a form that have yet to have any input entered.

> You shouldn't rely on the placeholder text to convey any essential information, as screen readers do not treat placeholder text as a label. Read more about that here: `https://www.w3.org/WAI/tutorials/forms/instructions/#placeholder-text`.
>
> On the subject of accessibility, ensure that you use appropriate contrast for the placeholder. If you don't already have a tool to check acceptable contrast levels, I'd recommend bookmarking `https://webaim.org/resources/contrastchecker/`.

The required attribute

We can also easily define when an input is required with the `required` attribute.

`required` is a Boolean attribute, with "Boolean" meaning it has only two possibilities. In this case, you either include it or do not.

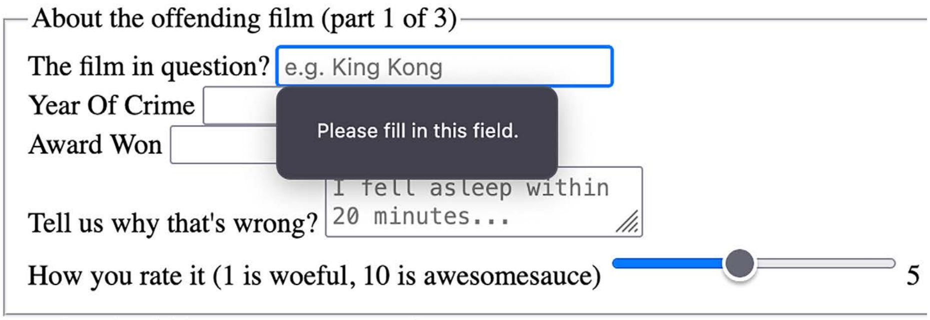
Here is an example of when it is present:

```
<input type="text" value="" placeholder="hal@2001.com" required />
```

It means that adding a value to this input will be required before the form can be submitted.

If the form submission is attempted without the input containing a requisite value, a warning message is displayed. The message itself differs both in content and styling, depending on the browser and the input type used. To be absolutely clear, the bad news here is that you can't change the message that the browser displays. But the good news is that it will provide the correct language version of the message for your users to match their language preference.

We've already seen what the `required` field browser message looks like in Chrome. The following screenshot shows the same message in Firefox:



About the offending film (part 1 of 3)

The film in question? e.g. King Kong

Year Of Crime

Award Won

Please fill in this field.

Tell us why that's wrong? I fell asleep within 20 minutes...

How you rate it (1 is woeful, 10 is awesomesauce) 5

Figure 14.3: Errors when submitting a form in Firefox without the required fields completed

The `required` value can be used alongside many input types to ensure that a value is entered. Notable exceptions are the `range`, `color`, `button`, and `hidden` input types, as they almost always have a default value.

The autofocus attribute

The HTML5 `autofocus` attribute allows a form to have a field focused, ready for user input, as soon as the page loads. The following code is an example of an input field wrapped in a `div` with the `autofocus` attribute added at the end:

```
<div>
  <label for="search">Search the site...</label>
  <input
    id="search"
    name="search"
    type="search"
    placeholder="Wyatt Earp"
    autofocus
  />
</div>
```

Tread carefully with `autofocus` for a few reasons.

Firstly, if multiple fields have `autofocus` added, only the first autofocused field will be focused. It's not hard to imagine a scenario where you accidentally added `autofocus` to one field, and some time later, added `autofocus` to another. Your user may not get the experience you expected, and you might have yourself a bug to troubleshoot.

Secondly, it's also worth considering that some users use the spacebar to quickly skip down the content of a web page. On a page where a form has an autofocused input field, this capability is negated; instead, a space is added to the focused input field. It's easy to see how that could be a source of frustration for users.

In addition, users of assistive technology will be instantly transported to a location on the page they have no control over—not exactly the best user experience!

If using the `autofocus` attribute, be certain it's only used once in a form, and be sure you understand the implications.

The autocomplete attribute

By default, most browsers aid user input by autocompleting the value of form fields where possible.

While the user can turn this preference on and off within the browser, we can now also indicate to the browser when we don't want a form or field to allow autocompletion with the `autocomplete` attribute. This is useful not just for sensitive data (bank account numbers, for example) but also if you want to ensure that users pay attention and enter something manually.

For example, for many forms I complete, if a telephone number is required, I enter a "spoof" telephone number. I know I'm not the only one who does that (doesn't everyone?), but I can ensure that users don't enter an autocompleted spoof number by setting the value of the `autocomplete` attribute to `off` on the relevant input field. The following is a code example of a field with the `autocomplete` attribute set to `off`:

```
<div>
  <label for="tel">Telephone (so we can berate you
    if you're wrong)</label>
  <input
    id="tel"
    name="tel"
    type="tel"
    placeholder="1-234-546758"
    autocomplete="off"
    required
  />
</div>
```

It's not possible to stop autocompletion on entire fieldsets in one go, but you can stop autocompletion on entire forms. Just add the `autocomplete` attribute to the `form` element itself. Here's an example:

```
<form id="redemption" method="post" autocomplete="off">
  <!-- content -->
</form>
```

Be aware that users of assistive technology may end up frustrated if turning off `autocomplete`. It can hinder shortcuts and prevent the identification of fields, leading to frustrated users.

The list attribute and the associated datalist element

This `list` attribute and the associated `datalist` element allow a number of selections to be presented to a user once they start entering a value in the field. The following is a code example of the `list` attribute in use with an associated `datalist`, all wrapped in a `div`:

```
<div>
  <label for="awardWon">Award Won</label>
  <input id="awardWon" name="awardWon" type="text" list="awards" />
  <datalist id="awards">
    <select>
      <option value="Best Picture"></option>
      <option value="Best Director"></option>
      <option value="Best Adapted Screenplay"></option>
      <option value="Best Original Screenplay"></option>
    </select>
  </datalist>
</div>
```

`datalist` contains the list of possible values for the `input`. To connect the `datalist` to the `input`, you have to set the value of the `list` attribute on the `input` to the `id` of the `datalist`.

In our example, we have added an `id` of `"awards"` to the `datalist` element and then set the value of the `list` attribute on the `input` to that.

Although wrapping the options with a `select` element isn't strictly necessary, it helps when applying scripts to add comparable functionality for browsers that haven't implemented the feature.

With our `list` and `datalist` wired up, the input field still appears initially to be just a normal text input field. However, when typing in the input, a selection box appears below it with matching results from the `datalist`. In the following screenshot, we can see the `list` in action (Firefox).

In this instance, as *B* is present in all options within the `datalist`, all the values are shown for the user to select from:

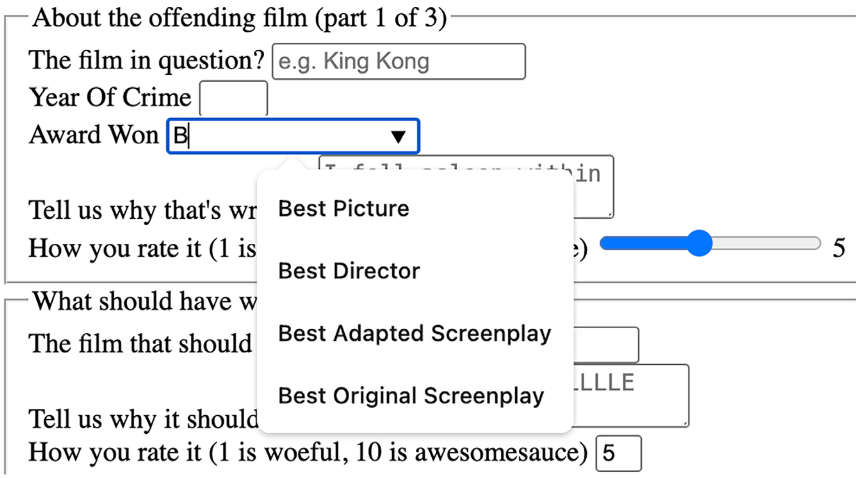

About the offending film (part 1 of 3)
The film in question? e.g. King Kong
Year Of Crime
Award Won B
Tell us why that's wr
How you rate it (1 is
What should have w
The film that should
Tell us why it should
How you rate it (1 is woeful, 10 is awesomesauce) 5
Best Picture
Best Director
Best Adapted Screenplay
Best Original Screenplay
5
LLLE

Figure 14.4: The datalist element showing the matching possible choices

However, when typing `D` instead, only the matching suggestions appear, as shown in the following screenshot:

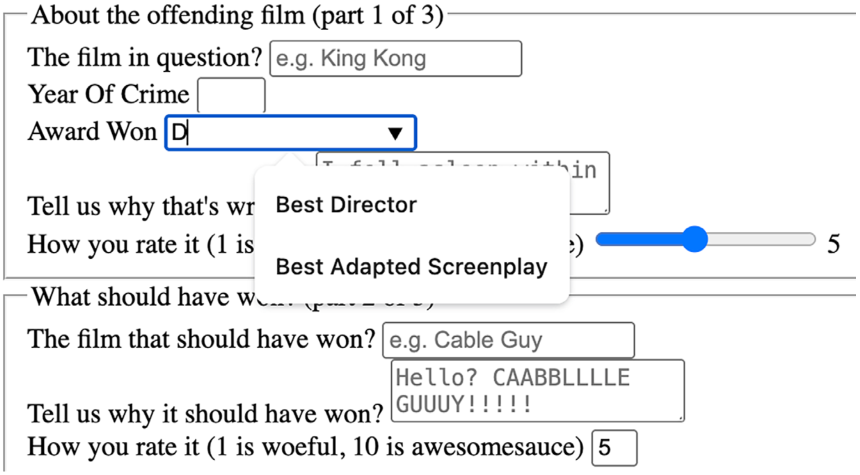

About the offending film (part 1 of 3)
The film in question? e.g. King Kong
Year Of Crime
Award Won D
Tell us why that's wr
How you rate it (1 is
What should have w
The film that should have won? e.g. Cable Guy
Tell us why it should have won? Hello? CAABBLLLLE GUUUY!!!!!
How you rate it (1 is woeful, 10 is awesomesauce) 5
Best Director
Best Adapted Screenplay
5

Figure 14.5: The datalist narrows based on input

`list` and `datalist` don't prevent a user from entering different text in the input box, but they do provide another great way of adding common functionality and user enhancement through HTML5 markup alone.

You can read the specification for `datalist` here: `https://html.spec.whatwg.org/multipage/form-elements.html#the-datalist-element`.

The pattern input attribute

You can set an input to expect a certain pattern input like this:

```
pattern=""
```

Note that this isn't an input type. However, it is a means of communicating to the browser that we expect input of a certain pattern.

The `pattern` attribute allows you to specify, via a **regular expression**, the syntax of data that should be allowed in a given input field.

Learn about regular expressions (regexes)

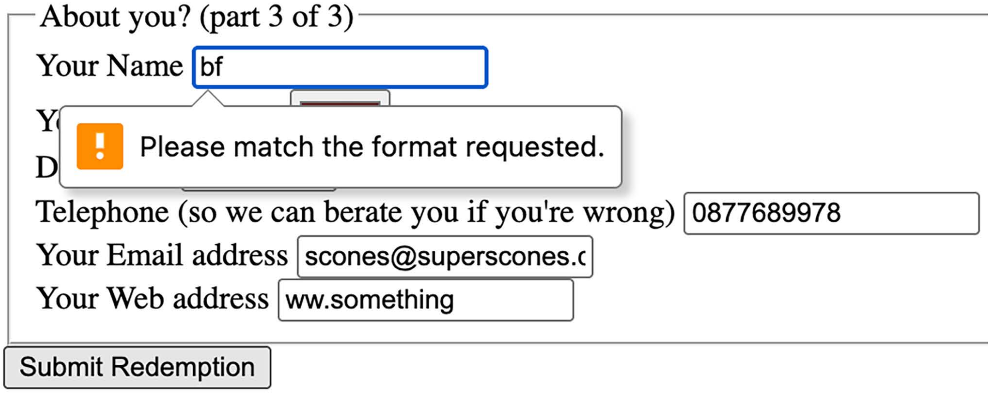
If you've never encountered regular expressions before, I'd suggest starting here: `https://en.wikipedia.org/wiki/Regular_expressions`. Regular expressions are used across many programming languages as a means of matching strings. While the format is intimidating at first, they are incredibly powerful and flexible. For example, you could build a regular expression to match a password format or select a certain style of CSS class-naming convention. To help build up your own regex pattern and get a visual understanding of how they work, I'd recommend starting with a browser-based tool such as `https://www.regexr.com/`. These days, it's very easy to cheat instead and just ask an AI, "Make me a regex for JavaScript and CSS that…".

The following code is an example:

```
<div>
  <label for="name">Your Name (first and last)</label>
  <input
    id="name"
    name="name"
    pattern="^([\D]{2,30}\s+)+([a-zA-Z]{2,30})$"
    placeholder="Dwight Schultz"
    required
  />
</div>
```

Such is my commitment to this book, I searched the internet for an entire 458 seconds to find a regular expression that would match a basic first and last name syntax (Western European languages only, sorry). This is by no means bulletproof, but it should ensure that the value entered is not a number (sorry, R2D2, you will have to register your film complaints elsewhere) and is made of at least 2 space-separated values between 2 and 30 characters long.

By entering the regular expression value within the `pattern` attribute, it makes supporting browsers expect a matching input syntax. Then, when used in conjunction with the `required` attribute, incorrect entries get the following treatment in supporting browsers. In this instance, I tried submitting the form without providing a last name:


About you? (part 3 of 3)

Your Name: bf

Please match the format requested.

Telephone (so we can berate you if you're wrong): 0877689978

Your Email address: scones@superscones.c

Your Web address: ww.something

Figure 14.6: The pattern type provides a vaguer warning but allows you to create bespoke requirements

As with other browser-generated form messages, it's not possible to amend it, so depending on your use case, it may be beneficial to state in your label text the kind of input you require.

Styling the input caret with the caret-color CSS property

Caret, which is pronounced "carrot" (like your favorite orange vegetable), in our context, refers to the insertion point in an input area. You might think of that typically blinking vertical line as a "cursor," but it is purposefully named differently in CSS to make the distinction from other cursors, such as the one produced by mouse input.

The `caret-color` property is now well supported and allows us to change the color of the caret. There are additional caret-related properties, such as `caret-animation` and `caret-shape`, but sadly, at the time of writing, they don't have any browser implementations, so currently, we are limited to changing just the color.

Suppose we wanted an orange input insert point; we could style it like this:

```
.my-Input {
  caret-color: #f90;
}
```

> In case you weren't aware, there is an attribute called `contenteditable` that can make the contents of an everyday element, such as a `div` or `span`, editable. You can make use of `caret-color` in those situations, too.

When we started this chapter, I mentioned that there are ways of hinting to the user and, depending on the device, aiding the user in entering the appropriate data for the input at hand. We can do that with HTML5 input types. Let me show you how.

HTML5 input types

HTML5 has a number of extra input types. These have been a great addition because when they are supported, they offer additional functionality, and when not supported, they do nothing, and the input continues to behave like a standard text type input.

> You should also be aware of the `inputmode` attribute that can also be used on inputs. As both the `inputmode` attribute and input type can affect the kind of software keyboard shown to the user on mobile devices, it is worth understanding the distinction. The `inputmode` attribute concerns itself with user experience, and crucially, can be used on an element in `contenteditable` mode. You might use it to suggest a numerical keypad to the user with `inputmode="numeric"`, or a keypad with decimal points by using `inputmode="decimal"`. However, unlike inputs, it does not require that the entered content conform to a particular data type, as we shall see that the input types do. Personally, I have rarely opted for `inputmode`, especially when dealing with forms, as the correct input type sets expectations of the content, and also hints as to the appropriate software keyboard to show.

Let's take a look at them.

The email input type

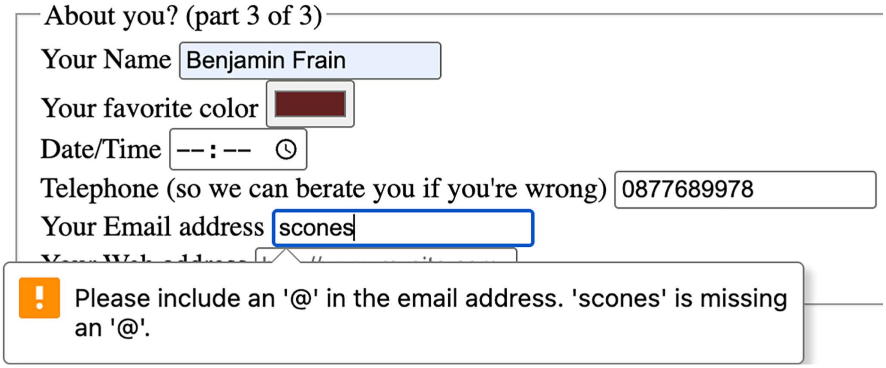
You can set an input to the `email` type, like this:

```
type="email"
```

Supporting browsers will expect a user input that matches the syntax of an email address. In the following code example, `type="email"` is used alongside `required` and `placeholder`:

```
<div>
  <label for="email">Your Email address</label>
  <input
    type="email"
    id="email"
    name="email"
    placeholder="dwight.schultz@gmail.com"
    required
    autocomplete="off"
  />
</div>
```

When used in conjunction with `required`, trying to input a non-conforming value will generate a warning message:



About you? (part 3 of 3)

Your Name: Benjamin Frain

Your favorite color

Date/Time: --:--

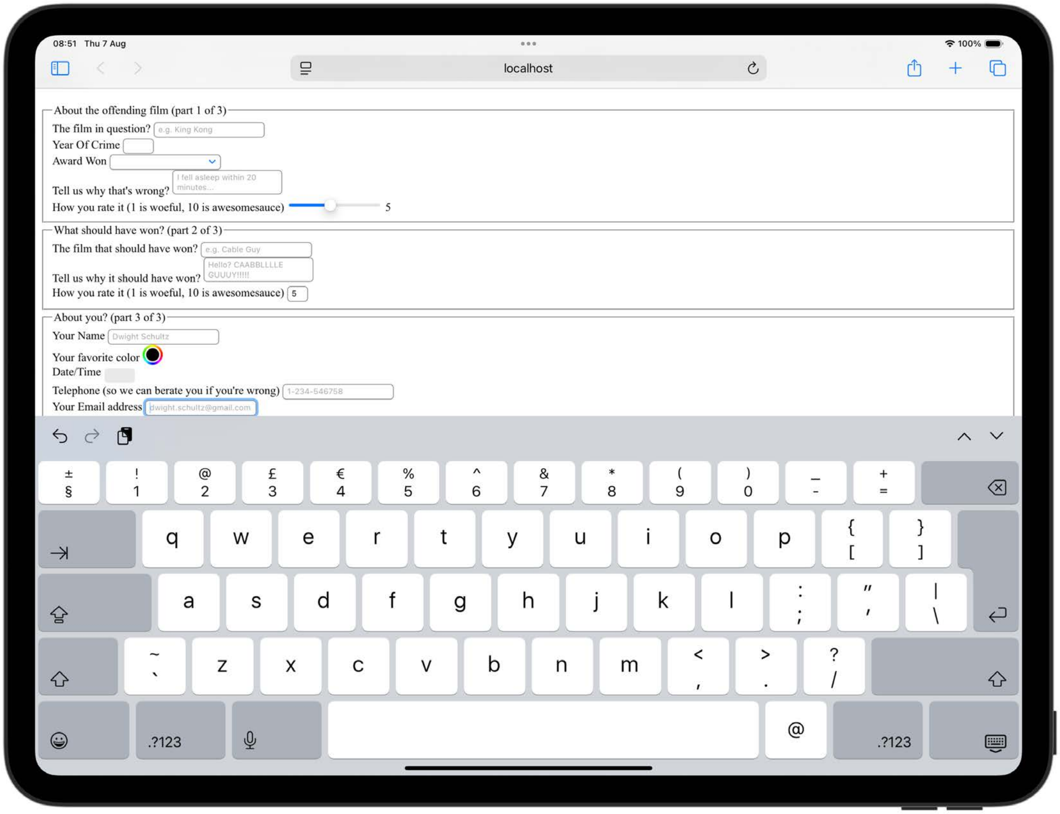
Telephone (so we can berate you if you're wrong): 0877689978

Your Email address: scones

Please include an '@' in the email address. 'scones' is missing an '@'.

Figure 14.7: An error occurs when incorrect data is entered

Perhaps most usefully, most touchscreen devices (Android, iPhone, and so on) change the software keyboard presented to the user based on this input type. The following screenshot shows how the software keyboard on an iPad is shown when focusing an input with `type="email"`. Notice the `@` symbol has been added for easy email address completion:



About the offending film (part 1 of 3)
The film in question? e.g. King Kong
Year Of Crime
Award Won
Tell us why that's wrong? I fell asleep within 20 minutes...
How you rate it (1 is woeful, 10 is awesomesauce) 5

What should have won? (part 2 of 3)
The film that should have won? e.g. Cable Guy
Tell us why it should have won? Hello? CAABBLLLLE GUUUY!!!!!
How you rate it (1 is woeful, 10 is awesomesauce) 5

About you? (part 3 of 3)
Your Name Dwight Schultz
Your favorite color
Date/Time
Telephone (so we can berate you if you're wrong) 1-234-546758
Your Email address dwight.schultz@gmail.com

Figure 14.8: Software keyboards will often adapt to the input type

The number input type

You can set an input field to expect a number, like this:

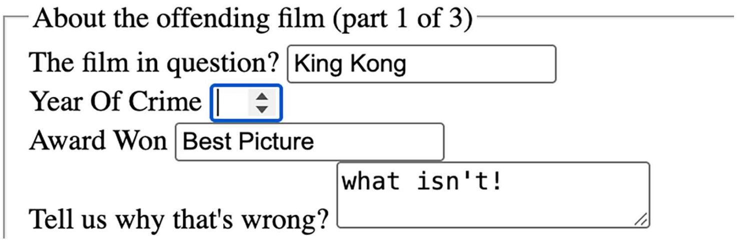
```
type="number"
```

With the type of input set to `number`, browsers also sometimes add pieces of UI called “spinner” controls. These are tiny pieces of user interface that allow users to easily click up or down to alter the value input.

The following is a code example:

```
<div>
  <label for="yearOfCrime">Year Of Crime</label>
  <input
    id="yearOfCrime"
    name="yearOfCrime"
    type="number"
    min="1929"
    max="2027"
    required
    step="10"
  />
</div>
```

The following screenshot shows how it looks in Chrome, complete with spinners—the UI presented to the user by the browser to aid them in decrementing/incrementing the number:



About the offending film (part 1 of 3)

The film in question? King Kong

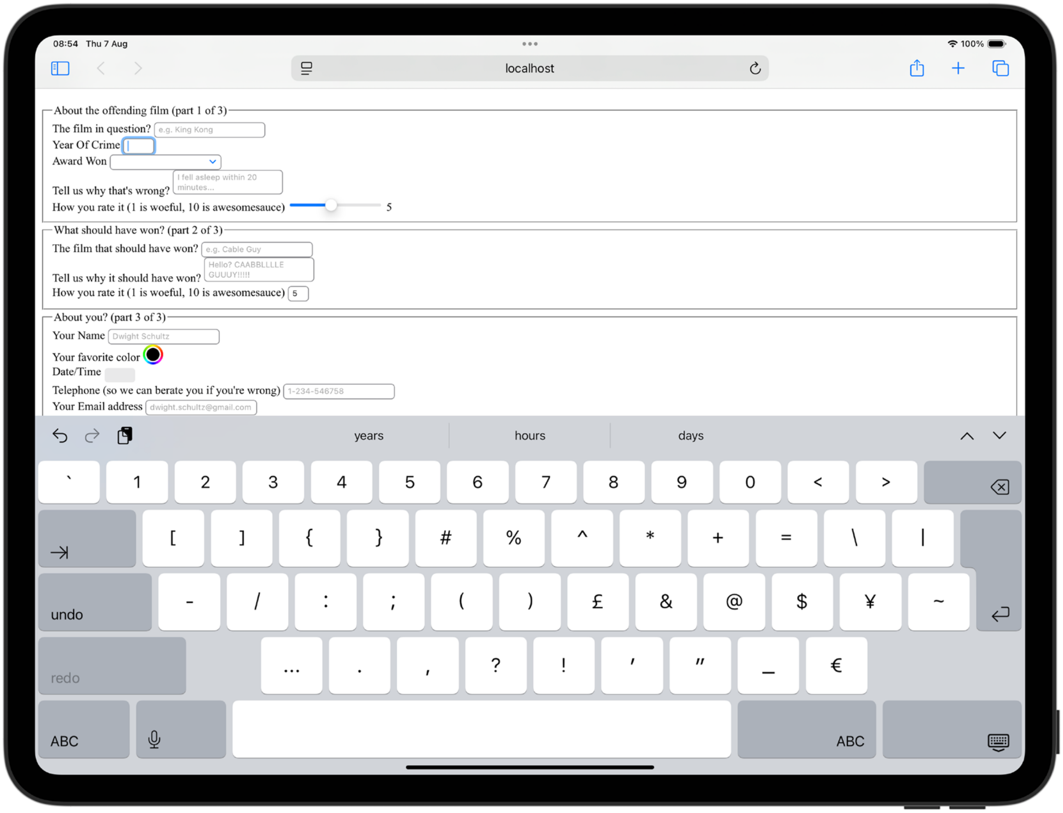
Year Of Crime

Award Won Best Picture

Tell us why that's wrong? what isn't!

Figure 14.9: On desktop browsers, “spinners” are shown for number inputs

And here is how `type="number"` makes the software keyboard appear on an iPad. Notice how all the numerical keys display by default:



About the offending film (part 1 of 3)
The film in question? e.g. King Kong
Year Of Crime
Award Won
Tell us why that's wrong? I fell asleep within 20 minutes...
How you rate it (1 is woeful, 10 is awesomesauce) 5
What should have won? (part 2 of 3)
The film that should have won? e.g. Cable Guy
Tell us why it should have won? Hello? CAABBLLLLE GUUUY!!!!!
How you rate it (1 is woeful, 10 is awesomesauce) 5
About you? (part 3 of 3)
Your Name Dwight Schultz
Your favorite color
Date/Time
Telephone (so we can berate you if you're wrong) 1-234-546758
Your Email address dwight.schultz@gmail.com

Figure 14.10: Software keyboards default to showing numbers for number inputs

What happens if you don't enter a number? This can vary subtly between browser implementations. For example, Firefox does nothing until the form is submitted, at which point it displays a warning, **Please enter a number**. Safari, on the other hand, gives the vaguer message of **Fill out this field**.

Using min and max to create number ranges

You'll notice in the previous code example that we set a minimum and maximum allowed range:

```
type="number" min="1929" max="2015"
```

If you try to submit the form with numbers outside of this range, the browser will show a warning that the value should be within the specified range. In addition, you can use the `:in-range` pseudo-selector to style the value when it is in range. You can also use the `:out-of-range` pseudo-selector for when values are out of the defined range. The typical frustration I encounter is that if there is no value set for `input`, it does not evaluate to being out of range. Instead, I tend to use the `:invalid` pseudo-selector, which styles blank inputs and out-of-range inputs. So, we could add this:

```
input#yearOfCrime {
    &:in-range {
        background-color: rgb(from #739327 r g b / 20%);
    }
    &:invalid {
        background-color: rgb(from red r g b / 10%);
    }
}
```

And that provides a pale red color when the input is blank or out of range, and a pale green when in range.

Changing the step increments

You can alter the step increments (granularity) for the spinner controls of numerical input types with the use of the `step` attribute. For example, to step 10 units at a time, use this:

```
<input type="number" step="10" />
```

The url input type

You can set an input field to expect a URL like this:

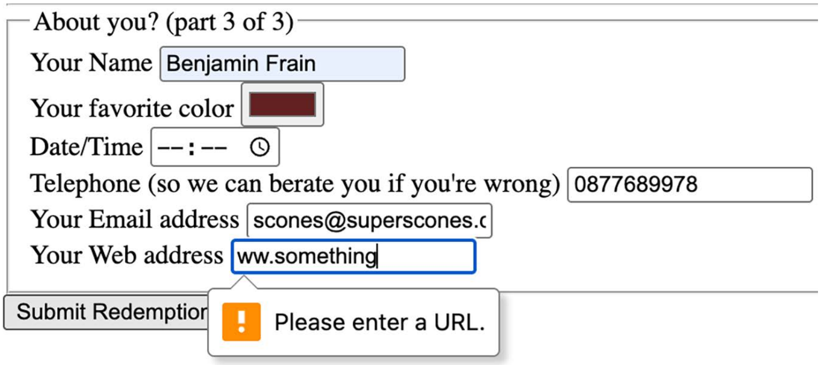
```
type="url"
```

As you might expect, the `url` input type is for URL values. Similar to the `tel` and `email` input types, it behaves almost identically to a standard text input. However, some browsers add specific information to the warning message provided when submitted with incorrect values. The following is a code example including the `placeholder` attribute:

```
<div>
  <label for="web">Your Web address</label>
  <input id="web" name="web" type="url"
    placeholder=https://www.mysite.com
```

```
      required
    />
</div>
```

The following screenshot shows what happens when an incorrectly entered URL field is submitted in Chrome:



About you? (part 3 of 3)

Your Name Benjamin Frain

Your favorite color

Date/Time --:--

Telephone (so we can berate you if you're wrong) 0877689978

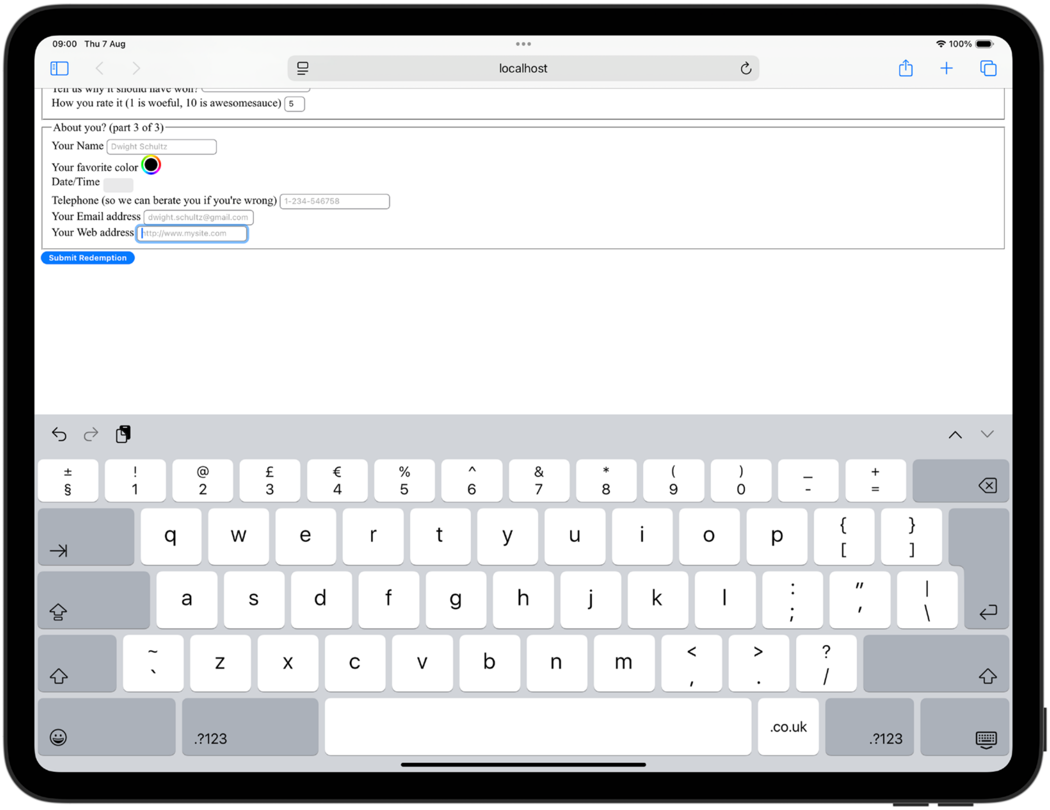
Your Email address scones@superscones.c

Your Web address ww.something

Please enter a URL.

Figure 14.11: Chrome will show a warning when the input doesn't match the type

Like `type="email"`, touchscreen devices often amend the software keyboard based on this input type. The following screenshot shows how the software keyboard of an iPad is changed with an input type set to `url`:



Tell us why it should have won?
How you rate it (1 is woeful, 10 is awesomesauce) 5

About you? (part 3 of 3)
Your Name Dwight Schultz
Your favorite color
Date/Time
Telephone (so we can berate you if you're wrong) 1-234-546758
Your Email address dwight.schultz@gmail.com
Your Web address http://www.mysite.com

Submit Redemption

Figure 14.12: Software keyboard adapted for URL input

Notice the *.co.uk* key next to the space bar? Because we've used a `url` input type, the software keyboard provides a key for easy top-level domain completion.

On iOS, if you're not going to a `.com` site, you can press and hold that button for a few other popular top-level domains.

Another occasionally useful attribute for inputs and `textarea` elements is `enterkeyhint`. This allows you to set it to the values of `enter`, `go`, `done`, `next`, `previous`, `next`, `search`, or `send`, and lets the UI update the label accordingly. I prefer to let the input type dictate that, but `enterkeyhint` is another tool for your collection!

The tel input type

You can set an input field to expect a telephone number type of value, like this:

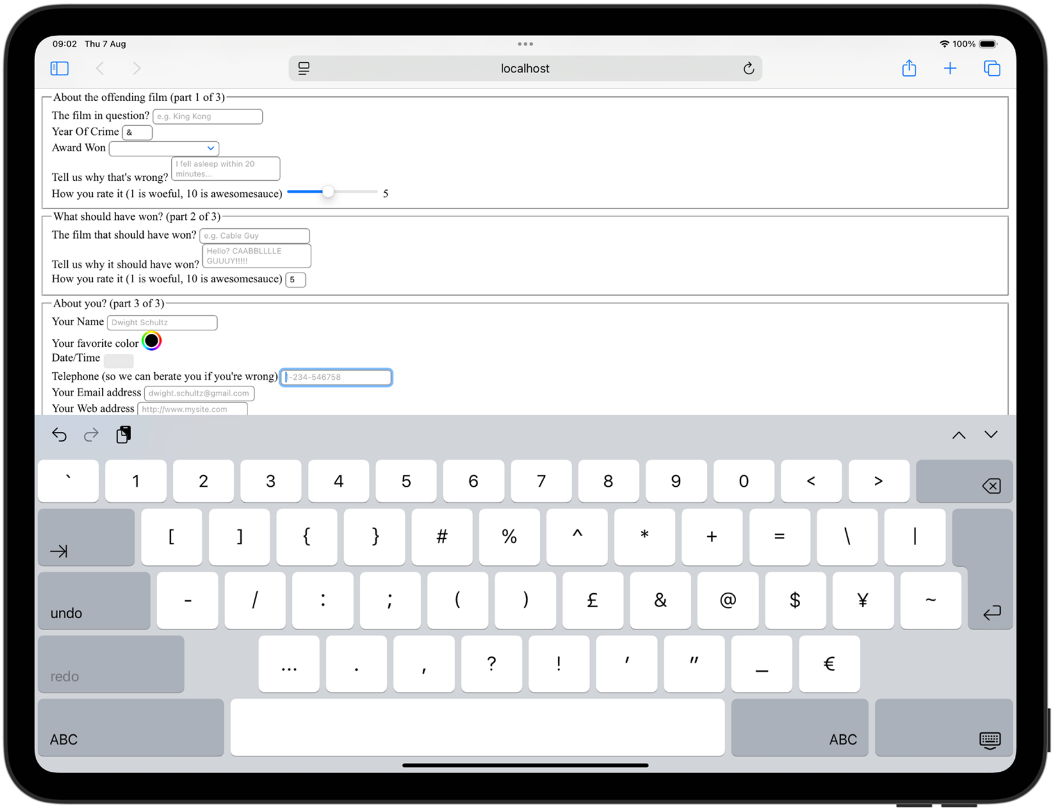
```
type="tel"
```

Here's a more complete example:

```
<div>
  <label for="tel">Telephone (so we can berate you
    if you're wrong)</label>
  <input
    id="tel"
    name="tel"
    type="tel"
    placeholder="1-234-546758"
    autocomplete="off"
    required
  />
</div>
```

Browsers do little validation on the `tel` input type. When an incorrect value is input, they fail to provide a suitable warning message.

However, some better news is that, like the `email` and `url` input types, touchscreen devices often thoughtfully accommodate this kind of input with an amended software keyboard for easy completion; here's the `tel` input type when accessed with an iPad:



About the offending film (part 1 of 3)
The film in question? e.g. King Kong
Year Of Crime &
Award Won
Tell us why that's wrong? I fell asleep within 20 minutes...
How you rate it (1 is woeful, 10 is awesomesauce) 5

What should have won? (part 2 of 3)
The film that should have won? e.g. Cable Guy
Tell us why it should have won? Hello? CAABBLLLLE GUUUY!!!!!
How you rate it (1 is woeful, 10 is awesomesauce) 5

About you? (part 3 of 3)
Your Name Dwight Schultz
Your favorite color
Date/Time
Telephone (so we can berate you if you're wrong) 1-234-546758
Your Email address dwight.schultz@gmail.com
Your Web address http://www.mysite.com

Figure 14.13: Software keyboard adapting to telephone input

Notice the lack of alphabet characters in the keyboard area? This makes it much faster for users to enter a value in the correct format.

If the default blue color of telephone numbers in iOS Safari annoys you when you use a `tel` input, you can amend it with the following selector:

```
a[href^=tel] { color: inherit; }
```

That will set them to the color of the parent element.

The search input type

You can set an input as a `search` type like this:

```
type="search"
```

The `search` input type works like a standard text input. Here's an example:

```
<div>
  <label for="search">Search the site...</label>
  <input id="search" name="search" type="search" placeholder="Wyatt Earp">
</div>
```

As with many of the prior input types, software keyboards (such as those found on mobile devices) often provide a more tailored keyboard. Note the *go* button in iOS when focused in a search box:

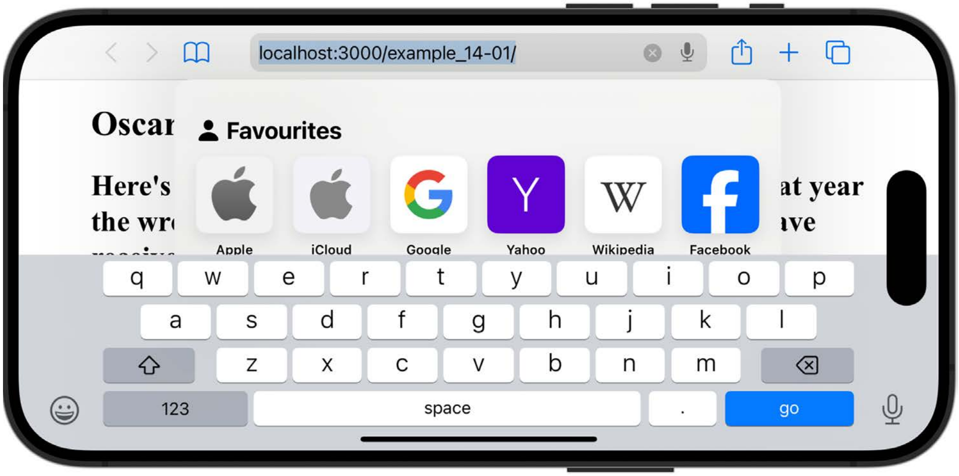

localhost:3000/example_14-01/
Favourites
Apple
iCloud
Google
Yahoo
Wikipedia
Facebook
123
space
go

Figure 14.14: A keyboard subtly tailored for searching

The color type input

Want to set an input field to receive a color value?

```
input type="color"
```

The `color` input type invokes the host operating system's color picker, allowing users to select a color. The following code is an example:

```
<div>
  <label for="color">Your favorite color</label>
  <input id="color" name="color" type="color" />
</div>
```

I'll be honest, it's not a type I've yet to need in practice, but it's not hard to imagine scenarios where it would be very handy.

Date and time

If you've ever bought tickets to an event online, chances are that you have used a date picker of one sort or another. The thinking behind the new `date` and `time` input types is so that the browser can provide a consistent piece of user interface for that situation.

Let's consider the core capabilities of the `date` and `time` input types.

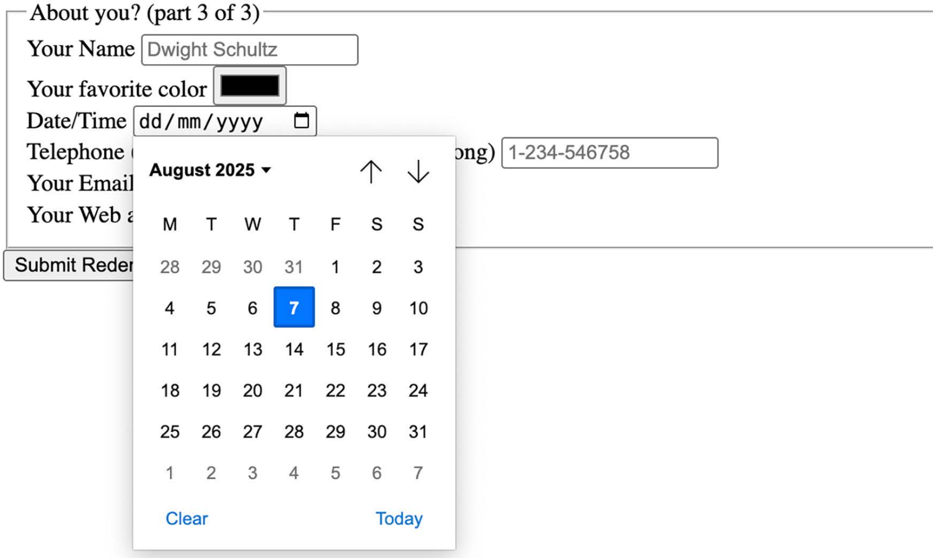
The date input type

The following code is an example:

```
<input id="date" type="date" name="date" value="2025-08-07"/>
```

Notice that it isn't just a matter of changing the `type` attribute; you might also want to use the `value` attribute to set the starting date for the picker. If you don't set your own date, the picker will default to today's date. If, however, you set a date in ISO8601 format (YYYY-MM-DD), inside the value, the start date will be set to that. Despite the value being written in code as YYYY-MM-DD, the browser will actually display the picker in a format better suited (arguably) to your own locale in the browser UI.

Here is the UI that is generated in a supporting browser. Note that as a UK user, I get that date displayed in a slightly more UK-friendly format (MM-DD-YYYY):



About you? (part 3 of 3)

Your Name Dwight Schultz

Your favorite color

Date/Time dd/mm/yyyy

Telephone (ong) 1-234-546758

Your Email

Your Web a

M	T	W	T	F	S	S
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31
1	2	3	4	5	6	7

Figure 14.15: A date-picking interface provided by the browser

However, depending on your locale, this isn't ideal and may actually be a worse experience for users. In the UK, we write dates in the DD-MM-YYYY format, so this could be confusing if you have a date such as 05-06-2024; a UK user would assume that was June 5, 2024. Your own locale may differ again. This discrepancy is what has led many institutions that deal with forms extensively to split the component parts of a date up into separate input fields. For example, the UK government's design system is well documented with examples—well worth a read if you encounter a form design heavy on the date side of things: `https://design-system.service.gov.uk/components/date-input/`.

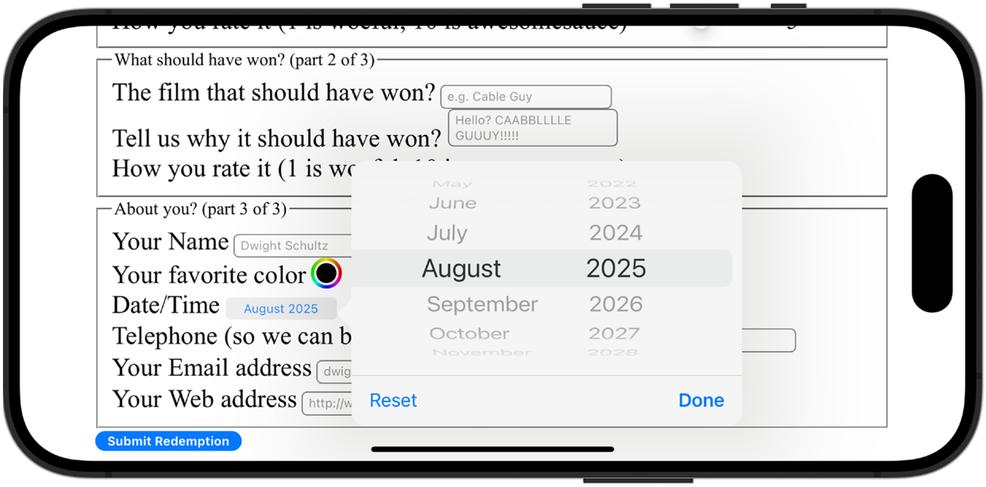
There are a variety of different date- and time-related input types available. What follows is a brief overview of the others.

The month input type

The following code is an example:

```
<input id="month" type="month" name="month" />
```

The interface allows the user to select a single month and sets the value of the input to a year and month, should you want to access it via scripting, for example, `2025-08`. Here is a grab of the user interface that iOS displays:



What should have won? (part 2 of 3)

The film that should have won? e.g. Cable Guy

Tell us why it should have won? Hello? CAABBLLLLE GUUUY!!!!!

About you? (part 3 of 3)

Your Name Dwight Schultz

Your favorite color

Date/Time August 2025

Your Email address

Your Web address

May 2022
June 2023
July 2024
August 2025
September 2026
October 2027
November 2028
Reset
Done
Submit Redemption

Figure 14.16: Remember, different operating systems often have their own UIs to achieve the same goal

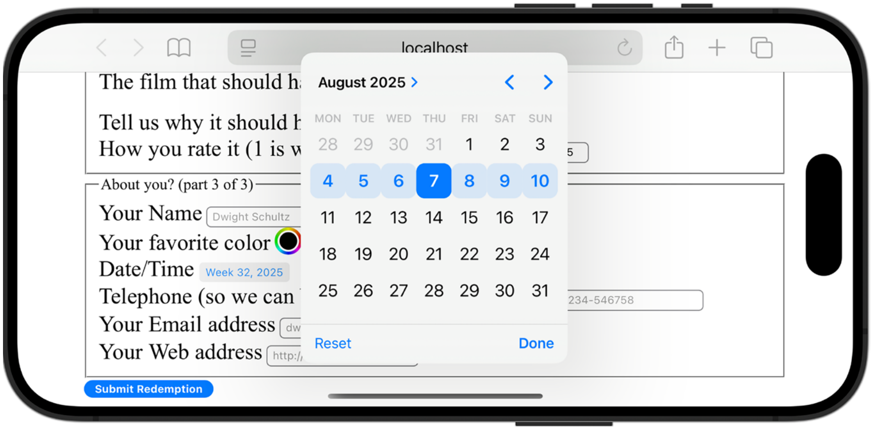
The week input type

The following code is an example:

```
<input id="week" type="week" name="week" />
```

When the `week` input type is used, the picker allows the user to select a single week within a year and sets the value of the input in `2025-W32` format.

The following screenshot shows what that looks like in iOS18:


localhost
August 2025
MON TUE WED THU FRI SAT SUN
28 29 30 31 1 2 3
4 5 6 7 8 9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
Reset
Done
Submit Redemption

The film that should h
Tell us why it should h
How you rate it (1 is w
About you? (part 3 of 3)
Your Name Dwight Schultz
Your favorite color
Date/Time Week 32, 2025
Telephone (so we can 234-546758
Your Email address dw
Your Web address http:/

Figure 14.17: The week input type gets its own data picker style in supporting browsers

The time input type

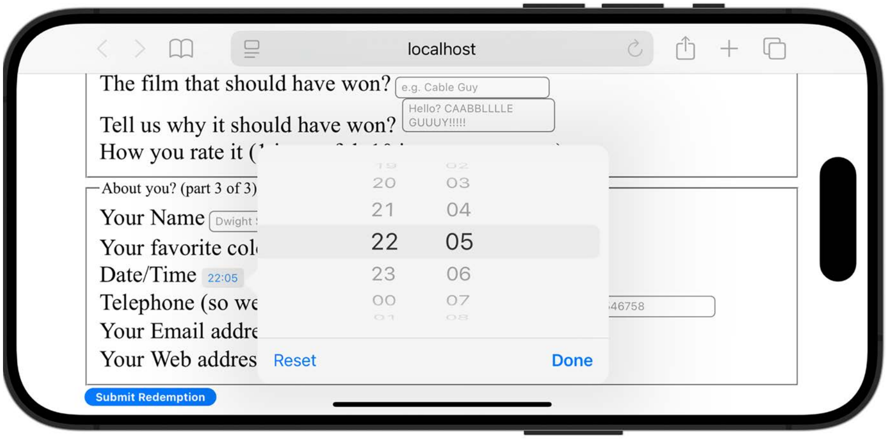
The following code is an example:

```
<input id="time" type="time" name="time" />
```

The `time` input type allows a value to be set to a 24-hour format, for example, `20:26`.

It displays like a standard input in supporting browsers, but with additional spinner controls, and it only allows relevant `time` values.

Touch devices show their own UIs. Here's how it looks on iOS:


localhost
20 03
21 04
22 05
23 06
00 07
Reset
Done
Submit Redemption

The film that should have won? e.g. Cable Guy
Tell us why it should have won? Hello? CAABBLLLLE GUUUY!!!!!
How you rate it (
About you? (part 3 of 3)
Your Name Dwight
Your favorite col
Date/Time 22:05
Telephone (so we 46758
Your Email addre
Your Web addres

Figure 14.18: The time input type produces specific pieces of interface in supporting browsers

The range input type

The `range` input type creates a slider interface element. Here's an example:

```
<input type="range" min="1" max="10" value="5" />
```

And the following screenshot shows how it looks in Firefox:

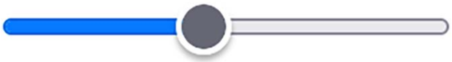


Figure 14.19: A range slider doesn't show numerical values by default

The default range is from `0` to `100`. However, by specifying a `min` and `max` value, in our example, we have limited it to between `1` and `10`.

One big problem I've encountered with the `range` input type is that the current value is never displayed to the user. Although the range slider is only intended for vague number selections, I've often wanted to display the value as it changes. Currently, there is no way to do this using HTML5. However, if you absolutely must display the current value of the slider, it can be achieved easily with some simple JavaScript. Amend the previous example to the following code:

```
<input
  id="howYouRateIt"
  name="howYouRateIt"
  type="range"
  min="1"
  max="10"
  value="5"
  onchange="showValue(this.value)"
/>
<span id="range">5</span>
```

We've added two things: an `onchange` attribute and a `span` element with the `id` value of `"range"`. Now, we'll add the following tiny piece of JavaScript:

```
<script>
  function showValue(newValue)
  {
    document.getElementById("range").innerHTML=newValue;
  }
</script>
```

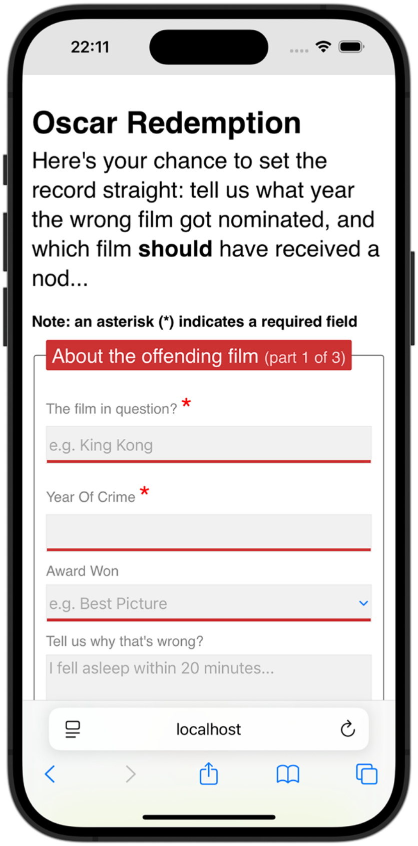
All this does is get the current value of the range slider and display it in the element with an `id` value of `"range"` (our `span` tag). You can then use whatever CSS you deem appropriate to change the appearance of the value. We will do that ourselves in a moment once we start styling our form.

Styling HTML5 forms with CSS

We have our HTML5-powered form built now, and understand the various input types and associated attributes. However, we need to make it a little more visually appealing across different viewport sizes. By applying some of the techniques we've learned throughout the previous chapters, I think we can improve the aesthetics of our form considerably.

I've taken my own stab at styling the form; you can check that version out at `example_14-02`, and remember, if you don't already have the example code, you can grab it at `https://rwd.education`.

Here's how the form looks in a small viewport with that basic styling applied:



Oscar Redemption

Here's your chance to set the record straight: tell us what year the wrong film got nominated, and which film **should** have received a nod...

Note: an asterisk (*) indicates a required field

About the offending film (part 1 of 3)

The film in question? *

e.g. King Kong

Year Of Crime *

Award Won

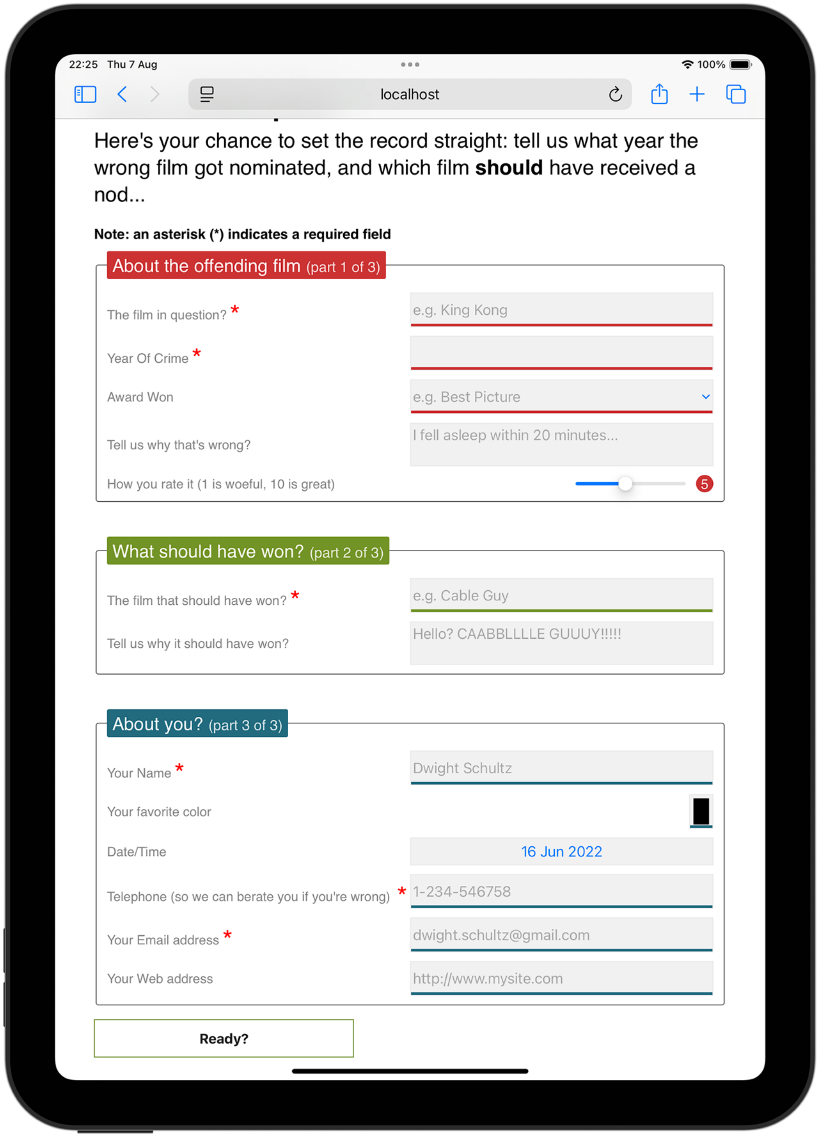
e.g. Best Picture

Tell us why that's wrong?

I fell asleep within 20 minutes...

Figure 14.20: Our form on mobile with basic styling applied

And here it is with a larger viewport:



Here's your chance to set the record straight: tell us what year the wrong film got nominated, and which film **should** have received a nod...

Note: an asterisk (*) indicates a required field

About the offending film (part 1 of 3)

The film in question? * e.g. King Kong

Year Of Crime *

Award Won e.g. Best Picture

Tell us why that's wrong? I fell asleep within 20 minutes...

How you rate it (1 is woeful, 10 is great) 5

What should have won? (part 2 of 3)

The film that should have won? * e.g. Cable Guy

Tell us why it should have won? Hello? CAABBLLLLE GUUUY!!!!!

About you? (part 3 of 3)

Your Name * Dwight Schultz

Your favorite color

Date/Time 16 Jun 2022

Telephone (so we can berate you if you're wrong) * 1-234-546758

Your Email address * dwight.schultz@gmail.com

Your Web address http://www.mysite.com

Figure 14.21: Our same form styled for wider viewports

If you look at the CSS in the example, you'll see many of the techniques we've looked at throughout previous chapters applied. For example, Flexbox (*Chapter 4*) is used to create uniform spacing and flexibility for elements. Transforms and transitions (*Chapter 11*) are used so that the focused input fields grow and the ready/submit button flips vertically when it gains focus.

Box shadows and gradients (*Chapter 8*) are used to emphasize different areas of the form. Media queries (*Chapter 3*) are used to switch the Flexbox direction for different viewport sizes, and more recent CSS selectors (*Chapter 6*) are used for selector negation.

But look, here's a thought. I'm no hotshot designer. How about you take the markup from `example_14-01` and have a shot at styling it up yourself? If it all goes horribly wrong, you can always continue with the styled version, but this would be a solid exercise to put into practice the considerable styling skills you have learned so far.

With that in mind, I won't be going over the techniques I chose in detail here again. Instead, I just want to focus on a couple of form-specific peculiarities—firstly, how to visually indicate required fields (and for bonus points, indicate a value has been entered) and, secondly, how to create a "fill" effect when a field gets user focus.

Indicating required fields

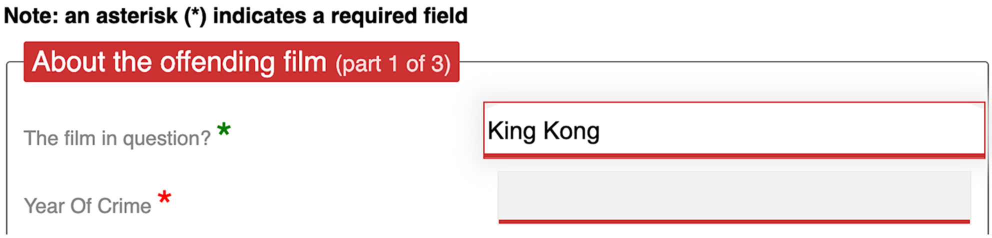
We can indicate required input fields to a user using CSS alone. Here is an example:

```
input:required {
  /* styles */
}
```

With that selector, we could add a border or outline to the required fields or add a `background-image` inside the field. Basically, the sky's the limit! We could also use a specific selector to target an input field that is required, but only when it gains focus. Here is an example:

```
input:focus:required {
  /* styles */
}
```

However, that would apply styles to the input itself. What if we want to amend styles on the associated `label` element? I've decided I'd like to indicate required fields with a little asterisk symbol at the side of the label:



Note: an asterisk (*) indicates a required field

About the offending film (part 1 of 3)

The film in question? * King Kong

Year Of Crime *

Figure 14.22: An asterisk to indicate a required field is a common pattern

Depending on how old the browser is that you need to support, this can present a problem. Historically, CSS only lets us effect a change on elements if they are children of an element, the element itself, or a general or adjacent sibling of an element that receives "state" (when I say *state*, I'm talking about `hover`, `focus`, `active`, `checked`, and so on). In the following examples, I'm using `hover`, but that would obviously be problematic for touch-based devices:

```
.item:hover .item-child {
}
```

With the preceding selector, styles are applied to `item-child` when the item is hovered over. Now, consider the following:

```
.item:hover ~ .item-general-sibling {
}
```

With this selector, when the item is hovered over, styles are applied to `item-general-sibling` if it is at the same DOM level as `item` and follows it. Next up, we have the following:

```
.item:hover + .item-adjacent-sibling {
}
```

Here, when the item is hovered over, styles are applied to `item-adjacent-sibling` if it is the adjacent sibling element of `item` (straight after it in the DOM).

So, back to our issue. If we have a form with labels and fields like this, with `label` above `input` (to give us the requisite basic layout), it leaves us a little stuck:

```
<div class="form-Input_Wrapper">
  <label for="film">The film in question?</label>
  <input
    id="film"
    name="film"
    type="text"
    placeholder="e.g. King Kong"
    required
  />
</div>
```

In this situation, until the CSS `:has` selector, there was no way to change the style of the `label` based on whether `input` is required or not (as it comes after `label` in the markup). So, let's look at two ways to solve the issue. The code for both is in the `example_14-02` files.

In recent browsers, which support the `:has` selector, which we looked at in *Chapter 6*, *CSS Nesting, Layers, Selectors, and More*, we can easily use a selector like this:

```
.form-Input_Wrapper:has(input:required) label::after {}
```

And then we can style the `::after` pseudo-element accordingly. This looks at the element that contains the label and says, "If I have a required input in here, select the label inside."

But if you need to support browsers that do not support the `:has` selector, let's consider how to work around the problem for older browsers.

We could switch the order of those two elements in the markup, which would then enable us to use an adjacent sibling selector to target it, but then we would end up with `label` displayed underneath `input`.

However, have you remembered that both Flexbox and Grid give us the ability to visually reverse the order of elements (read all about them in *Chapter 4* and *Chapter 5* if you haven't done so already) with ease?

That allows us to use the following markup to create a direct sibling association we can use in CSS:

```
<div class="form-Input_Wrapper">
  <input
    id="film"
    name="film"
    type="text"
    placeholder="e.g. King Kong"
    required
  />
  <label for="film">The film in question?</label>
</div>
```

And then we can simply apply `flex-direction: row-reverse` or `flex-direction: column-reverse` to the parent (or use a template and areas in the case of Grid). These declarations reverse the visual order of their child elements, allowing the desired aesthetic of the label above (smaller viewports) or to the left (larger viewports) of the input.

Now, we can get on with actually providing some indication of required fields and when they have received input.

Thanks to our revised markup, the adjacent sibling selector now makes this possible:

```
input:required + label:after {
}
```

This selector essentially says, “For every label that follows an input with a `required` attribute, apply the enclosed rules.” Here is the CSS for that section:

```
input:required + label:after {
    content: "*";
    font-size: 2.1em;
    position: absolute;
    top: -8px;
    display: inline-flex;
    margin-left: 0.2ch;
    transition: color 1s;
}

input:required:invalid + label:after {
    color: red;
}

input:required:valid + label:after {
    color: green;
}
```

Then, if you focus on a required input and enter a relevant value, the asterisk changes color to green. It’s a subtle but helpful touch.

We are using valid and invalid pseudo-selectors here to change the color of the text. You should also be aware that there is also `:user-valid` and `:user-invalid`. These differ in that they only evaluate once a user has interacted with the element or tries to submit the form. These two may be more suitable selectors to use, depending on your use case.

> There are more form-specific pseudo-selectors than we have covered in this chapter that may be able to help you out, depending on the problem you are trying to solve, such as `:default`, `:checked`, and `:focus-within`. For a full list of pseudo-selectors, head over to this section of the specification: `https://drafts.csswg.org/selectors/#useraction-pseudos`.

At this point, it seems appropriate to tell you that the `content` value is not read out by all screen readers, which makes the point once more that using the right elements, labels, and attributes is always the best policy. In this case, the presence of the `required` attribute should provide a screen reader with the relevant prompt to communicate this need to the user.

Creating a background fill effect

Back in *Chapter 9*, *Stunning Aesthetics with CSS*, we learned how to generate linear and radial gradients as background images. Sadly, it isn't possible to transition between two background images (which makes sense, as the browser effectively rasterizes the declaration into an image). However, we can transition between values of associated properties such as `background-position` and `background-size`. We'll use this capability to create a fill effect when an `input` or `textarea` receives focus.

Here are the properties and values added to `input`:

```
input:not([type='range']),
textarea {
  min-height: 40px;
  padding: 4px;
  font-size: 17px;
  border: 1px solid #ebebeb;
  outline: none;
  transition: transform 0.2s, box-shadow 0.2s, background-position 0.2s;
  background: radial-gradient(circle at center 400px, #fff 99%,
    transparent 99%),
    #f1f1f1;
  background-position: -400px 90px, 0 0;
  background-repeat: no-repeat, no-repeat;
  border-radius: 0;
  position: relative;
}
input:not([type='range']):focus,
textarea:focus {
  background-position: 0 0, 0 0;
}
```

In the first rule, a solid white radial gradient is being generated but positioned out of view. The background color that sits behind (the hex value after `radial-gradient`) is not offset and so provides a default color. When `input` gains focus, `background-position` on `radial-gradient` is set back to the default, and because we have `transition` on `background-image` set, we get a nice transition between the two. The visual result of this is the appearance of the input being “filled” with a different color when it gains focus.

It’s an effect better used than viewed as a screenshot, so be sure to try out the example code.

New <select> and <option> styling

There is no denying that styling form elements to a designer’s exact whim can be challenging. They are arguably the most opinionated elements on a web page, with each browser and OS determining how certain elements can and cannot be displayed.

There has, however, been a push among browser makers of late to bring more customizability to form elements. The first fruit of that labor is the ability to create a customized select menu. However, you should be aware that as I write this in late 2025, there is only an implementation in Chromium browsers, such as Chrome and Edge. But thanks to progressive enhancement, it isn’t broken on other browsers, you just get something that works like a normal select menu.

Let me show you how it works.

Default select styling

Let’s define the problem. Suppose I am making a form, part of which requires a user to choose their nationality. My HTML might look like this:

```html
<select name="lang" id="standard-select">
    <option value="">Choose your country...</option>
    <option value="argentina">Argentina</option>
    <option value="brazil">Brazil</option>
    <option value="canada">Canada</option>
    <option value="denmark">Denmark</option>
    <option value="equador">Equador</option>
    <option value="germany">Germany</option>
</select>
```

And by default, that would look like this:

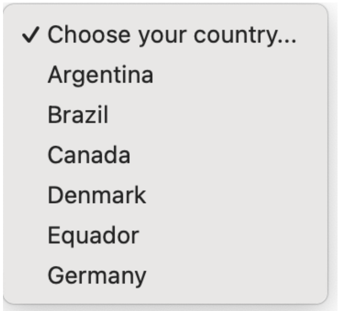

✓ Choose your country...
Argentina
Brazil
Canada
Denmark
Equador
Germany

Figure 14.23: The default select menu

Not exactly the most exciting thing in the world, is it? What I would like to do is add a flag next to each country option to help users easily identify their country. And this is typical of the kind of thing developers have been asked to do with select menus for years. Until recently, achieving a visually enhanced menu, such as this, had required all manner of workarounds, and typically different HTML elements *pretending* to be a select menu with all the accessibility issues that entails! But now we can use customizable selects in HTML and get this:

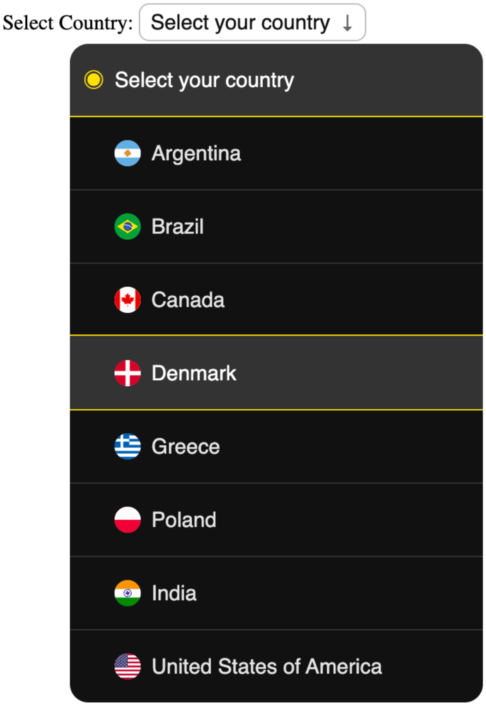

Select Country:
Select your country ↓
Select your country
Argentina
Brazil
Canada
Denmark
Greece
Poland
India
United States of America

Figure 14.24: Customized select menu and options

Let's look at what we need to do to achieve this.

Customizable selects

Let's start with the markup. You can find this in the code samples as `example_14-03`. I won't copy the entire markup out, as each option has the same structure. Here are the pertinent parts:

```
<select id="lang-select">
    <button>
        <selectedcontent></selectedcontent>
    </button>

    <option class="option-slat" value="">Select your country</option>
    <option class="option-slat" value="argentina">
        <img
            class="flag"
            src="./flags/Argentina.svg"
            alt="Flag of Argentina"
        />
        <span class="option-name">Argentina</span>
    </option>
```

We have a `select` element, and the first child is a `button`. The first oddity is the `selectedcontent` element inside the button. That's a new thing in HTML5, and this essentially clones the selected option from the list of options inside itself.

Then, each option can include normal HTML markup. I've chosen to use an `img` for the flag of the country, but you don't have to. You might go with `span`, `div`, or whatever is needed to style your option.

Don't worry, if a browser does not know anything about these new capabilities, it will simply ignore them. This is progressive enhancement in action!

If you use this markup at this point, the `select` menu dropdown will work exactly as a normal `select` menu. To give it the new visual abilities, we need to opt into that, and we do that in CSS:

```
select,
::picker(select) {
    appearance: base-select;
}
```

With that in place, and nothing more, if you refresh, the menu will display with the flags showing alongside the names of the counties.

The `base-select` value for the `appearance` property also provides a new minimal look. That's good news because the new minimal styles are simple to override with whatever you want.

There are a few new selectors and pseudo-selectors that come along with this new customizable select. `::picker(select)` is the dropdown itself, `selectedcontent` is the element inside the button that launches the menu, and `select::picker-icon` is the little graphic to indicate there is a menu. I opted to use a down arrow emoji for the one in the example, but it could be anything that you could put in the `content` of any other pseudo-element.

I'd encourage you to take a look inside the CSS of the example code to get a feel for what's possible. You can see the menu itself is positioned using an `anchor()` function. What is this hot new sauce I hear you cry? Don't worry, we will be looking at `anchor()` and plenty of other new shizzle (my teenager tells me that word means *stuff*, he's not lying, is he?) in *Chapter 15*, *Cutting-Edge CSS Features*.

> There are a few things that this customizable `select` does miss out on compared to the standard one. Chiefly, it can no longer render outside the browser window, and it doesn't trigger the OS's default UI for the `select` element (remember how, on a mobile browser, you get the OS's own UI for choosing `select` options).

Summary

In this chapter, we learned how to use a host of HTML5 form input types and attributes. They have enabled us to make forms more usable and helpful, and in turn, allow us to capture better quality information.

We've also made use of some techniques we have learned throughout this book to restyle our form and make its layout respond to the constraints of the device on which it is used.

We're nearing the end of our responsive HTML5 and CSS journey. But before we get to our final chapter of parting advice, there are a bunch of bleeding-edge features I'm excited to share with you in the next chapter. Some of these have just shipped in browsers as I write this, but they are so powerful that I think they are worth looking at, even without cross-browser implementations to test them all on.

You got 30 minutes? If you have been doing web development for years, some of this will blow your mind!

Part 3

Latest Platform Features and Parting Advice

In the first part of the book, we covered the essentials of responsive web design. We didn't just cover the principles; we took deep dives into media queries and modern layout systems such as Flexbox and Grid.

In *Part 2*, we concentrated on the CSS features I believe are essential for solving all manner of day-to-day styling challenges. We covered selecting items, typography, new color features, SVG, animations, and heaps more; the sort of tools that you will be using day in and day out if your work involves CSS. And frankly, if your work doesn't involve CSS, I'm not sure why you are here! Mom, is that you?

In the next two chapters, which comprise the final part of this book, I want to cover a couple of broader topics: a bunch of tips, tricks, and techniques that didn't necessarily fit elsewhere, and some of the latest platform features.

15

Cutting-Edge CSS Features

Depending on when you are reading this, the contents of this chapter may seem commonplace and less "cutting-edge" in your daily CSS writing. However, as these words are being written, these are some of the latest additions to the CSS language.

The good news is, there are implementations of all the features we will look at in major browsers such as Chrome and Safari. The only caveat to you using them will be how old the versions of each browser are that you need to support.

Regardless, these are settled APIs that are part of the web platform.

So, here is what we will take a look at in this chapter:

- The Popover API
- Anchor positioning
- Scroll-driven animations
- View transitions

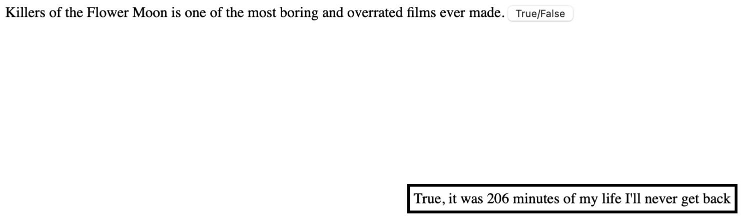
The Popover API

As you might guess, the Popover API allows you to declaratively "pop" a piece of content "over" another.

This is incredibly useful because, with few exceptions, ordinarily, if you want some UI to do something when clicked, you would need JavaScript. You can still use JavaScript with the Popover API, but you can also just use HTML. Consider this code:

```
<p>
    Is 'Killers of the Flower Moon' one of the most boring and
        overrated films ever made?
    <button popovertarget="answer">Yes/No</button>
    <span id="answer" popover>
        Yes! It was 206 minutes of my life I'll never get back
    </span>
</p>
```

That's `example_15-01`, and if you open it in a browser and click the **Yes/No** button, you will get something that looks like this.



Killers of the Flower Moon is one of the most boring and overrated films ever made.

True, it was 206 minutes of my life I'll never get back

Figure 15.1: By default, the popover toggles and sits in the center of the viewport

Let's examine the mechanics here. That `span` element, which is our popover, has an attribute of `popover`. It is set to `display: none` by default by the browser. The `button` invokes the popover by using a `popovertarget` attribute set to be equal to the `id` value of the element we want as our popover, which is `answer` in this example.

This linking of the invoker, with the `popovertarget` attribute set to a value matching the `id` value of the popover itself, means that they don't need to be siblings in the DOM. When invoked, the popover is shown in the browser's "top layer," a place of special privilege that places it automatically "higher" than normal content.

The top layer is the same place that the browser renders the dialog element and such, which we looked at in *Chapter 2*, *Writing HTML Markup*. You can read a little more on the top layer in this blog post on the Chrome site: `https://developer.chrome.com/blog/what-is-the-top-layer`.

At this point, there is no additional styling applied. That center position and the black border around the popover are added by default by the browser. We will get to styling shortly, and then positioning, but first, there are some other things to consider, mechanics-wise.

Toggling is the default behavior, but you can also set a `button` to show, and another to hide, using the `popovertargetaction` attribute. You add that alongside the `popovertarget` attribute like this:

```
<button popovertarget="answer" popovertargetaction="show">Show</button>
<button popovertarget="answer" popovertargetaction="hide">Hide</button>
```

One `button` hides and the other shows. Both of those are also shown in `example_15-01`.

Styling the popover

So far, we haven't touched the styling of the popover. Let's amend our prior example and now look at `example_15-02`, where basic additional styling has been added. I'll point out the things specifically relevant to the popover as we go.

First of all, in terms of selecting any popovers, you can use a general attribute selector, `[popover]`, as we discussed in *Chapter 6*, *CSS Nesting, Layers, Selectors, and More*. That is perhaps useful for resetting the browser-applied styles to popovers, but little else. Let's do that first:

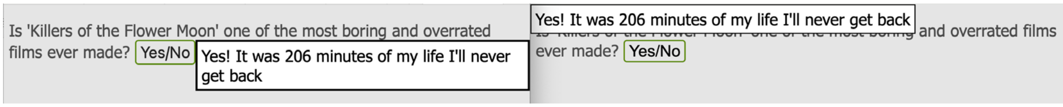
```
[popover] {
    inset: unset;
    margin: 0;
}
```

Those two declarations stop the popover from appearing in the center of the screen.

I have also added a darker wash to appear behind the popover when it is shown, to draw attention to the box. That was actually straightforward, as popovers also give us the `::backdrop` pseudo-element. That means we can add our darker color behind the popover like this:

```
.popover-content {
    &::backdrop {
        background-color: rgb(0 0 0 / 10%);
    }
}
```

But it's not all good news. If you open the popover in Safari and Chrome, you will find that it is in different positions.



Is 'Killers of the Flower Moon' one of the most boring and overrated films ever made? Yes/No

Yes! It was 206 minutes of my life I'll never get back

Yes! It was 206 minutes of my life I'll never get back

and overrated films ever made? Yes/No

Figure 15.2: Different renderings – Safari on the left, Chrome on the right

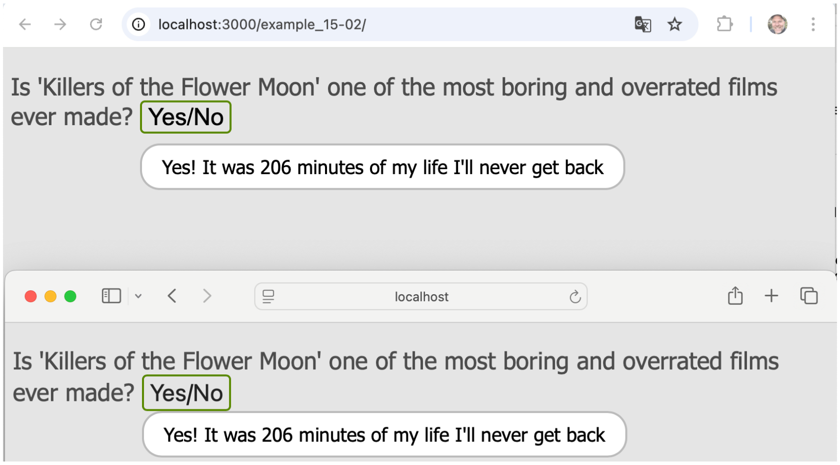
So we likely need to be a little more specific in our styling to achieve consistency. I would like my popover to behave like a tooltip, so that wherever that button is on the page, the popover opens next to it.

To achieve that, we are going to use another newer feature of CSS: anchor positioning. We are going to cover anchor positioning in detail in the next section. For now, just know that there is an automatic, or implied, anchor made between the invoker of the popover and the popover itself. That allows us to position our popover relative to the button that opens it.

I've added the following styles to the example file. There are some other arbitrary styles, but I have omitted them from the following code as they don't relate to the popover or its positioning specifically:

```
.popover-content {
    top: calc(10px + anchor(bottom));
    position-area: block-end span-inline-end;
    &::backdrop {
        background-color: rgb(0 0 0 / 10%);
    }
}
```

Don't worry about how those styles are doing what they are doing just yet; that will all be covered in the *Anchor positioning* section. All you do need to know for now is that with popovers, there is a built-in link between the invoker of the popover and the popover itself, and we can exploit that link to provide positioning to the popover content. And with that, the popover is opening in a consistent position on both Safari and Chrome.



Is 'Killers of the Flower Moon' one of the most boring and overrated films ever made? Yes/No

Yes! It was 206 minutes of my life I'll never get back

Is 'Killers of the Flower Moon' one of the most boring and overrated films ever made? Yes/No

Yes! It was 206 minutes of my life I'll never get back

Figure 15.3: With anchor positioning, the popover opens in a consistent position

There is also a `hint` value for popovers, but currently, it cannot be adopted across all browsers, so I've purposely refrained from covering it.

You can read the specification for the Popover API here: `https://html.spec.whatwg.org/multipage/popover.html`.

There are additional things to try with popovers. I'll leave it to you to take `example_15-02` further. What about animating the popover's entrance somehow using `@starting-style`, or using `transition-behavior` to add a transition as the popover goes from `display: none` to another value? We covered all that in *Chapter 12*, *Transitions, Transformations, and Animations*, so head back there if you need a refresher.

There are also `beforetoggle` and `toggle` JavaScript events so you can hook into when a popover is about to be shown, or has just been shown or hidden. Using these event listeners for popovers is not something I have needed yet, as I have only just started using them myself, but it's worth knowing they exist in case the need arises.

Aside from the basic mechanics of popovers, I think the most important thing to get a decent handle on is the positioning aspect of popovers. That means taking a detailed look at anchor positioning.

Anchor positioning

In nautical scenarios, an anchor is used to keep a boat tethered to a certain location, preventing it from moving against the natural flow of the sea. The current of the water might move the boat relative to the anchor a little, but it will always find itself equidistant from the anchor position.

CSS anchor positioning is a new module that provides an equivalent ability for elements on a web page – allowing elements to be connected to other elements, regardless of their structural relationship in the DOM.

If you have been tasked with building any semi-complicated UI on the web, you will have encountered instances where CSS anchor positioning would help. Think about tooltip boxes that pop up next to elements, warnings and information text associated with form fields, drop-down menus that span off navigation items, or even something as routine as positioning something in the corner of another, which ordinarily would necessitate one item being a child or sibling of the other, and then using `position: absolute` with some inset values to move it into position. The list goes on. Previously, we needed to add JavaScript to solve these issues. Now we can do it with CSS alone.

As anchor positioning is a new addition to CSS, it comes with new properties, values, and functions. The abundance of new properties and values for anchor positioning reminds me of CSS grid, in that there are many ways to achieve similar functionality but using different combinations of properties and values. Expect it to take a few runs at using it before the different mechanisms start to make sense.

Fundamentally, anchor positioning involves defining an element to behave as the anchor, and another element to be attached and positioned on the page relative to that anchor element.

We saw an implied anchor position relationship when we looked at the Popover API in the last section, where the popover is anchored to the invoking element. Anchor positioning lets you create explicit relationships between arbitrary elements, too. Perhaps when you are done reading this section, you can head back to the prior popover examples and try some of your newfound knowledge there?

First, let's deal with how we create an anchor positioning relationship between elements. You can follow along with this in `example_15-03`.

> If you want a fun way to play about with anchor positioning and test your understanding, I can recommend the Anchoreum game at `https://anchoreum.com/`.

Creating an anchor relationship

Right, at the risk of stretching this nautical anchor analogy to breaking point, here is what we have in our example at the outset:

```
<div class="sea"><div class="anchor"></div></div>
<div class="boat"></div>
```

And here is the CSS of those elements:

```
.sea {
    width: 800px;
    height: 400px;
    position: relative;
    opacity: 0.8;
    background-image: url("./assets/sea.jpg");
}

.anchor {
    background-image: url("./assets/anchor.png");
    background-size: contain;
    height: 80px;
    width: 80px;
    top: calc(50% - 40px);
    left: calc(50% - 40px);
    position: absolute;
}

.boat {
    background-image: url("./assets/boat.png");
    background-size: contain;
    aspect-ratio: 800 / 377;
    height: 80px;
}
```

And it looks like this.

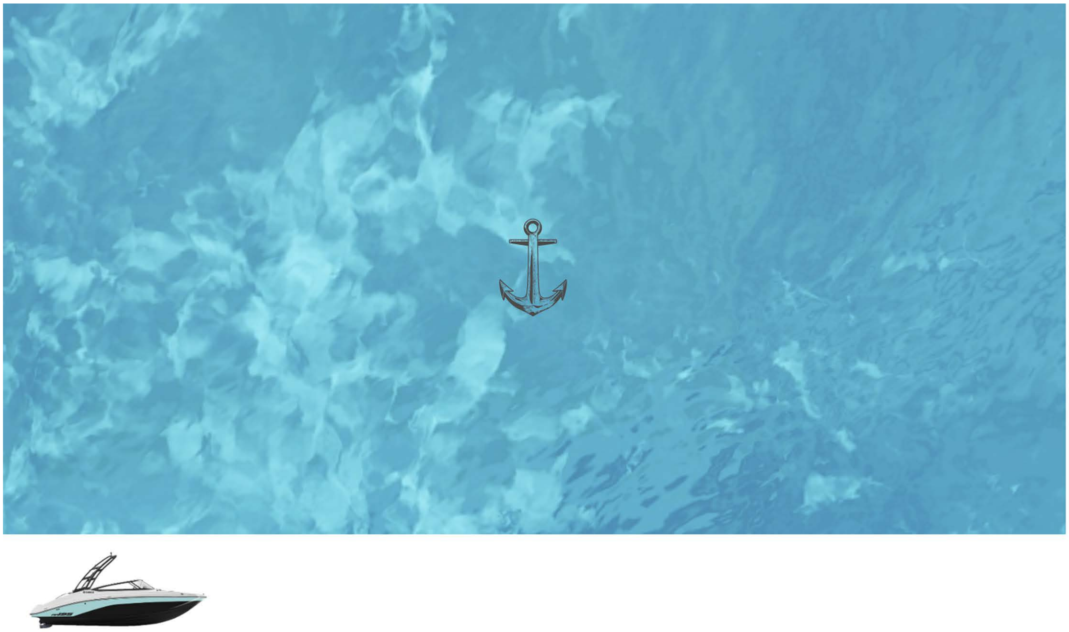


Figure 15.4: Before any anchor positioning, the boat is not in the sea!

The sea has an anchor in it, but at present, the boat is not even in the sea (I told you I would be stretching this analogy!).

DOM-wise, the anchor element is nested inside the sea element, but the `boat` element has no parent/child relationship to either. Let's create a relationship using anchor positioning. To do this, we need to give our anchor a name and tell the item to which we want to anchor it which anchor position to use.

So, on the anchor, we add `anchor-name`, using the same syntax we would use for a custom property – two dashes and then the name:

```
.anchor {
    /* other styles… /*
    anchor-name: --boat-anchor;
}
```

Then, on the element we want to be anchored to it, that is,`.boat`, we add the `position-anchor` property and value.

Now, if you were to take a look at the element in the DOM and start trying to use any of the `position-area` properties, as we will shortly, nothing would happen. The identifier for the `position-anchor` property, `--boat-anchor`, would even appear grayed out in the Chrome/Edge developer tools. So, here's the first important thing to remember with anchor positioning. The item that is being anchored needs to be set to either `position: fixed` or `position: absolute`. Both of those remove the element from the normal document flow and allow it to be positioned with anchor positioning. Let's set the `.boat`, like this:

```css
.boat {
    /*other styles /*
    position-anchor: --boat-anchor;
    position: absolute;
}
```

Positioning anchored content with position-area

Perhaps the most straightforward way to position the anchored element is with the `position-area` property.

Imagine the anchor element in the center of a three-by-three grid. The `position-area` property allows you to use a keyword to position your element in any of the areas. For example, a value of `right top` would position the anchor to the right-hand side and above the anchor:

```css
.boat {
    background-image: url("./assets/boat.png");
    background-size: contain;
    aspect-ratio: 800 / 377;
    height: 80px;
    position-anchor: --boat-anchor;
    position: absolute;
    position-area: right top;
}
```

> Aspect ratio can be useful when you want to apply a `background-image` to an element and keep it the same shape as you resize it. Because the image is 800x377 px, I can set that as the `aspect-ratio` value so it stays in proportion when the `height` value is adjusted.

Refer to the following figure. The values in the lighter color are the logical values; I prefer to use physical values, but just use the kind you prefer using.

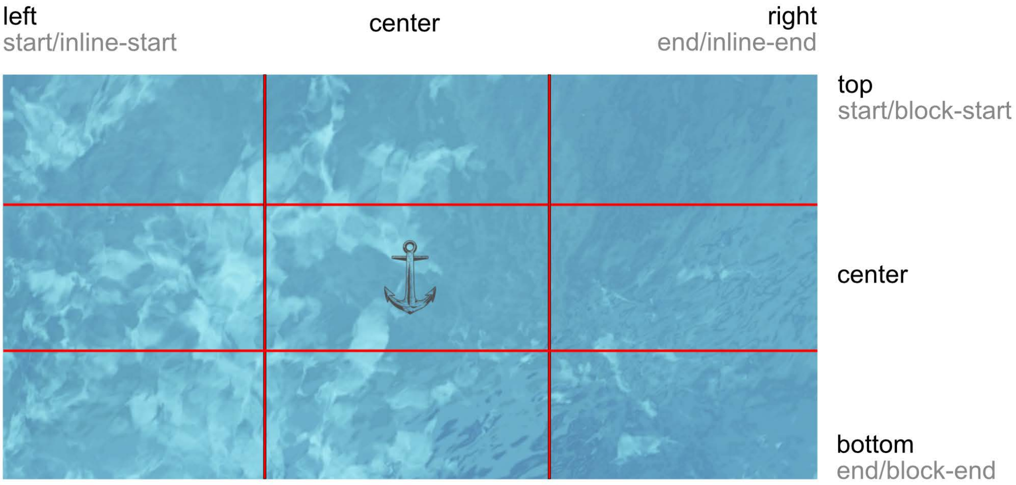


Figure 15.5: Using position-area keywords to position the element

You can also span across multiple areas of that grid. For example, if you wanted your anchored element to be below but span from the middle to the right, you could do `position-area: y-end span-x-end`. I prefer the *x*/*y* axis naming convention over the logical equivalent properties, such as `position-area: block-end span-inline-end`, but you can use either. You can also span all columns or rows with `span-all`. Ultimately, all these variations can accomplish similar results, with no one approach better than the other; just use the one that best suits your mental model.

Here is what we might do if our anchored area were 600 px wide, and we wanted it to span the bottom area, below the anchor:

```
.boat {
    background-image: url(./assets/boat.png);
    background-size: contain;
    height: 80px;
    position-anchor: --boat-anchor;
    position: absolute;
    position-area: y-end span-all;
    background-color: #e3cbcb;
    width: 600px;
}
```

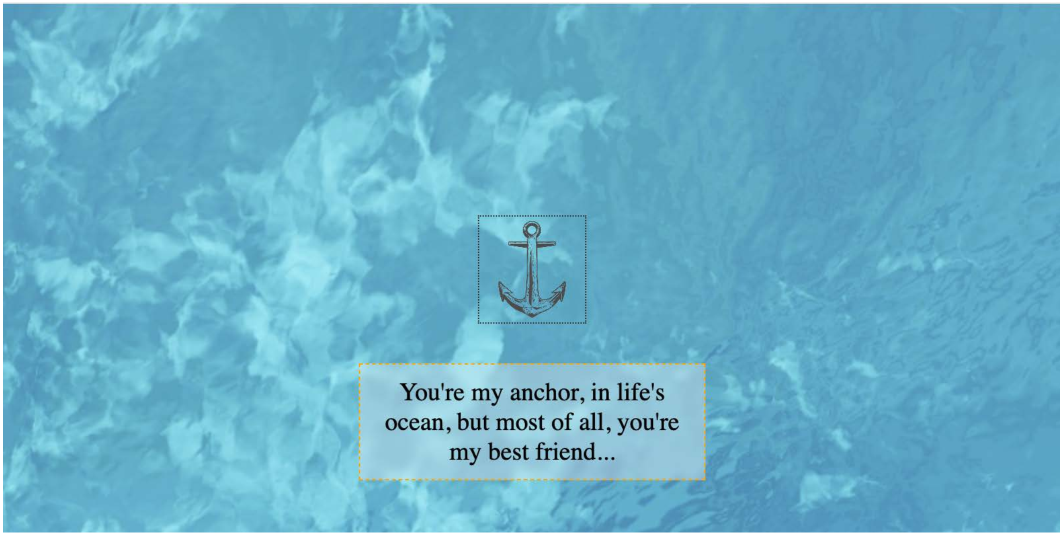
Positioning with anchor() functions

If one of the predefined areas, or spans of areas, does not suit your needs, fear not. You can get even more fine-grained control over positioning with `anchor()` functions.

The `anchor()` function can be used as a value for any of the `inset` properties (`top`, `bottom`, `left`, or `right`), which are computed relative to an edge of the anchor.

Take a look at `example_15-04`. Here, we are setting the anchor element exactly as we did in the prior example, but instead of the boat graphic, we have a box with an anchor-related song lyric in it.

If you open it in the browser, it should look like this:



You're my anchor, in life's ocean, but most of all, you're my best friend...

Figure 15.6: Fine-grained positioning with the anchor() function

We achieve that positioning like this. Note that I have omitted the declarations here that are not to do with positioning:

```
.text {
    position-anchor: --boat-anchor;
    position: absolute;
    inset: calc(anchor(bottom) + 30px) calc(anchor(right) - 90px) auto
        calc(anchor(left) - 90px);
}
```

So, what exactly is going on here? When we use `position: absolute` or `position: fixed` to position something, we can use the `inset` values to set the amount we want the edge of our element to be in relation to either the nearest non-static positioned element, in the case of `absolute`, or in relation to the viewport, as in the case of `fixed`.

The `anchor()` function resolves to a length, and so it allows us to inset in relation to the anchor position.

The `anchor()` function can take three arguments. The first is the anchor name, which is optional and only needed if you have multiple anchors on the page. Then there is the anchor side, which could be `center`, `left`, `right`, `top`, `bottom`, or their logical equivalents, or a percentage, which is the distance along the anchor in the related axis. That affords quite a bit of positioning flexibility! The final, optional argument for `anchor()`, is the fallback size. This is the percentage that will get used if the `anchor()` function cannot resolve to what it has been set to.

Frankly, I never bother with the fallback, and rarely bother with the anchor name either as I have found it unusual in practice to have multiple anchored elements on the same page.

In our example, we set the `inset` top value to be the bottom of the anchor plus 30 px. The right-hand inset was set to the right-hand side of the anchor plus 90 px, the bottom inset to `auto`, and the left `inset` value to the left-hand side of the anchor minus 90 px.

Sizing the anchored elements with anchor-size()

There's an anchor-related function I'll confess I have not found much use for yet, and that's `anchor-size()`. It allows you to size your anchored content in relation to the anchor's size.

`anchor-size` takes the same three arguments that `anchor()` does but instead of dealing with position, it deals with the *size* of the anchor. Practically, it allows you to perform a height/width measurement on an element to set the size of another. Again, this is something that in the past would have necessitated JavaScript but can now be done in CSS.

Remember the last example? Let us suppose we wanted to size our text as a multiple of the anchor size. So, we wanted it to be three times the width of the anchor, positioned at the right in the center, and about 10 px off the left edge of the anchor. It should look like this:

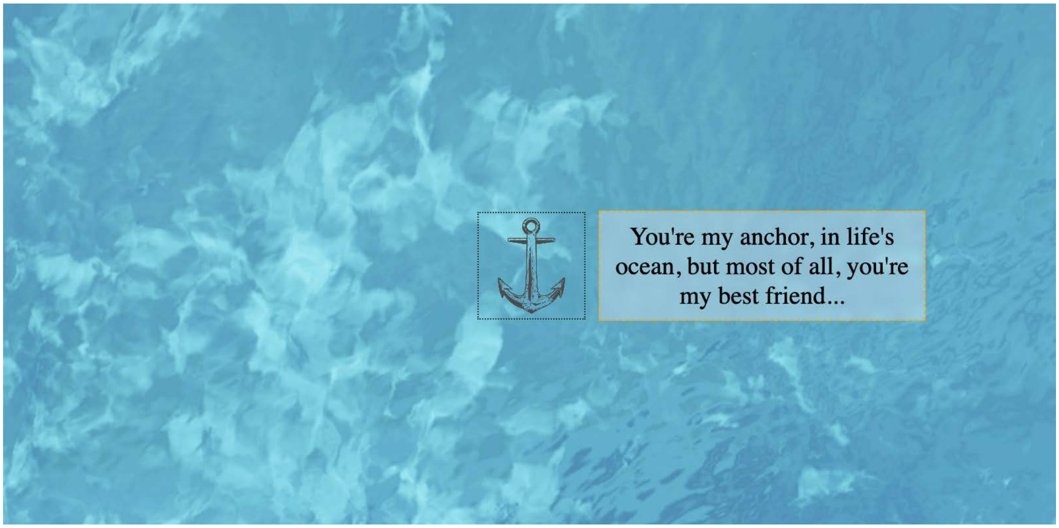

You're my anchor, in life's ocean, but most of all, you're my best friend...

Figure 15.7: Sizing elements based on their anchor with anchor-size()

Here is the code I used to achieve that. As before, I have omitted styles that aren't related to size and position, but take a look at them all and play about with this yourself in `example_15-05`:

```
.text {
    position-anchor: --boat-anchor;
    position: absolute;
    position-area: right center;
    left: 10px;
    width: calc(anchor-size(width) * 3);
    height: anchor-size(height);
}
```

Overflow management with position-try

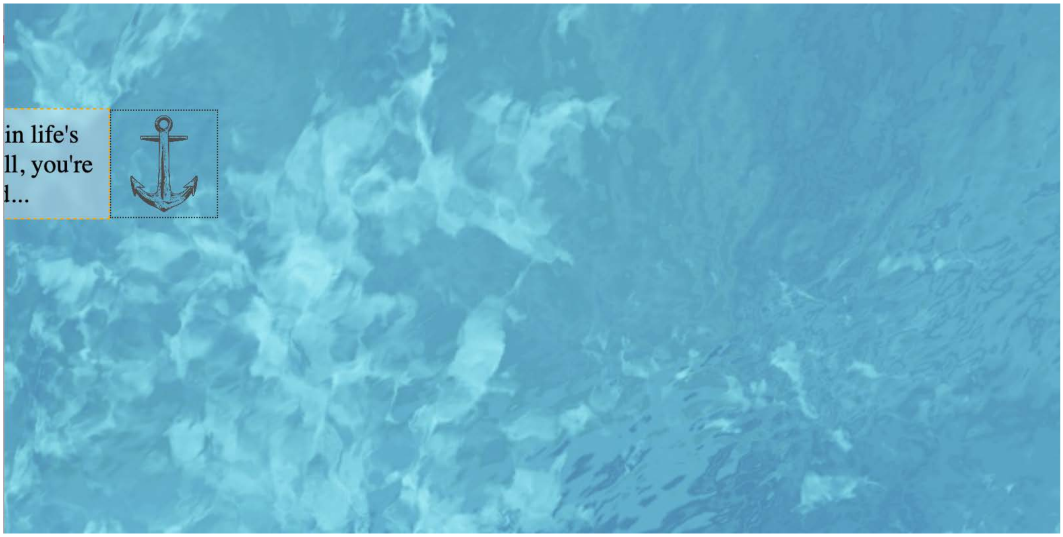
If you have ever tried to make any kind of tooltip popover content with JavaScript, you will have doubtlessly encountered a scenario where your popover appears but it is cut off by the viewport. Elements close to the edge of the screen are the most susceptible. Thankfully, the spec writers and implementors of the Popover API and Anchor Positioning API have considered this. As such, in the anchor positioning specification, we have a section dealing with overflow management that covers the `position-try` shorthand and the `@position-try` at-rule.

Using position-try

`position-try` is actually a shorthand property, allowing you to set `position-try-fallbacks` and `position-try-order` in one go.

Personally, I prefer using the shorthand to set both in one go, partly because it is terser to write, and partly because I rarely bother using `position-try-order`.

Consider `example_15-06`. Here, our anchor is near the left edge of the screen, and our anchored content has been set to three times the width of the anchor. Without any remedial overflow management, the anchored element would flow off the screen.



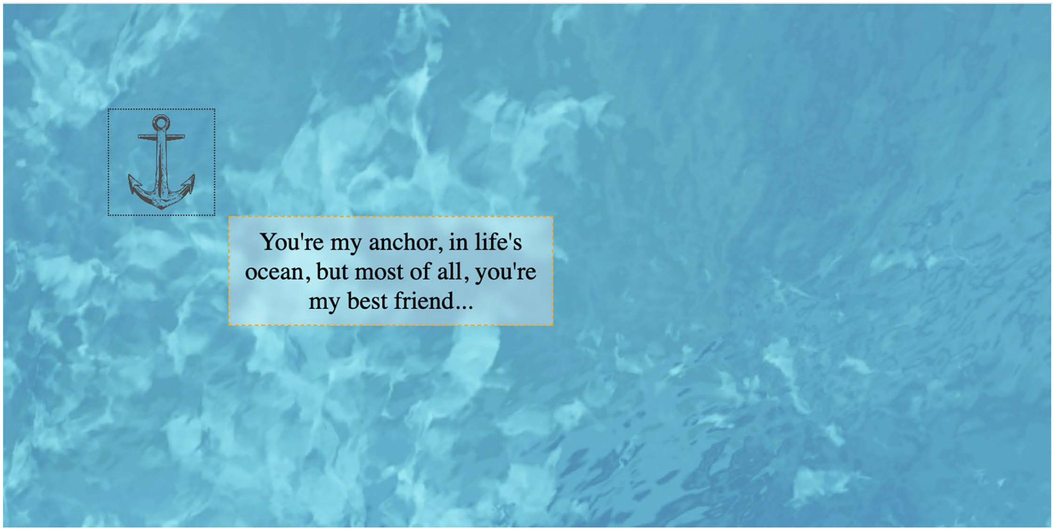
in life's
ll, you're
l...

Figure 15.8: position-try allows you to safeguard against overflow situations

Let’s fix that by adding this:

```
position-try: most-width right bottom, right;
```

Now, despite `anchor-position` of this element being set to `left center`, as the item is overflowing its containing box, the anchor-positioning algorithm has been asked to use the `most-width` “tactic” to choose the most relevant choice from the fallback options – and we have provided two options, `right bottom` and `right`, in that order.



You're my anchor, in life's ocean, but most of all, you're my best friend...

Figure 15.9: By providing options, the algorithm chooses an option where the content does not overflow

The reality is, for most practical situations, I'm not interested in giving a `try` tactic; I just give `position-try` a bunch of different `position-try-fallbacks` as a comma-separated list until it finds one where the item fits. In this case, I might just opt for the following:

```
position-try: right bottom, right;
```

In this scenario, that would have the same effect.

position-try flip tactics

Instead of specifying precise fallback options as positions to try, we can provide `position-try` (more accurately, the `position-try-fallbacks` part of the shorthand) with a "tactic" instead of a bunch of options. The tactics are conveyed with the `flip-block`, `flip-inline`, and `flip-start` keywords.

Both `flip-block` and `flip-inline` hopefully do what you expect, trying the opposite position in that axis where there isn't enough space. If you use `flip-start`, it swaps the start position of each axis, effectively trying the opposite position in the diagonal direction.

Hopefully, the tactics and options we can employ are starting to make some sense. There are a lot of options with anchor-positioning, so just as with something like grid, expect to take a few goes before it clicks.

One other possibility, which I generally find most useful, is defining one or more custom positions for anchoring to try, and we do that with the `@position-try` at-rule.

Custom positions with @position-try

We can combine `inset`, `margin`, and sizing properties into a `@position-try` rule. We can then pass as many of those `@position-try` rules as we like to the `position-try` property. Let's make a couple now:

```
@position-try --rule-one {
    position-area: right;
    width: 500px;
    top: 50%;
}

@position-try --rule-two {
    position-area: bottom right;
    width: 400px;
    top: 20px;
}
```

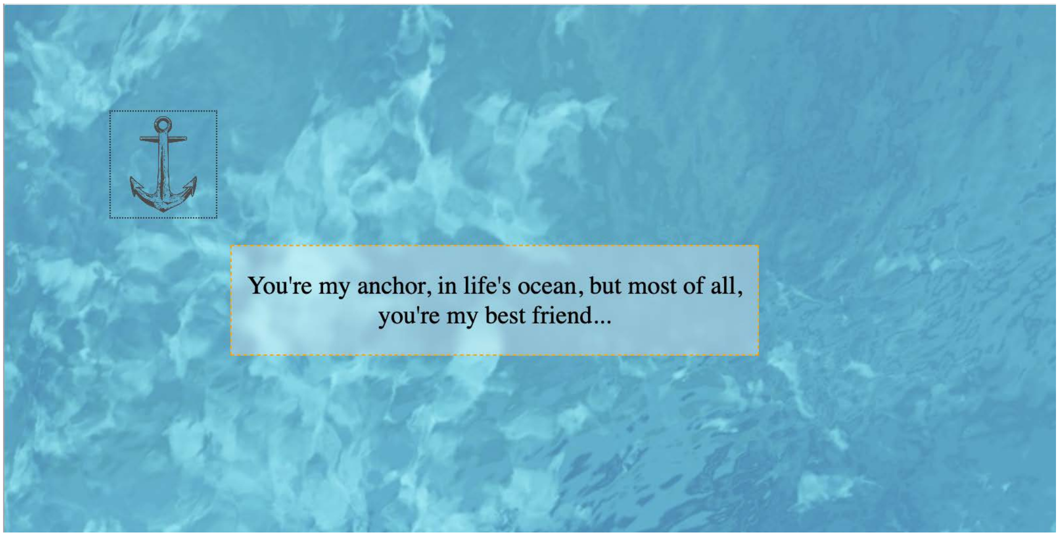
Each custom rule gets an ident, or name, in the common custom property format, which is to prefix the name with two dashes. Then, inside the braces are the positioning declarations. Remember, these can only be related to sizing, `inset`, and `margin`. Anything else in there will do nothing.

Then, we apply them to our element in `position-try` like this:

```
position-try: --rule-one, --rule-two;
```

Just like providing `position-area` options to `position-try`, it will try each in turn until the content no longer overflows. You can find this in `example_15-07`. This is a great option if you know exactly what you want and don't want the randomness that might occur when providing a tactic instead.

But the keen-eyed among you might be looking at that example and wondering why the browser chose `--rule-two` when it was second in the list, and the properties in `--rule-one` would also stop it overflowing. Turns out that the algorithm also takes into account which of the options would end up being the least deviation from the intended size and position. In this case, the `width` value of `--rule-two` is closer to the original computed `width` value of the anchored content. Pretty clever, huh?



You're my anchor, in life's ocean, but most of all, you're my best friend...

Figure 15.10: position-try will choose what best matches the intended dimensions

Wowsers, turns out anchor positioning can get pretty involved!

> There are more features and intricacies of anchor positioning. You can read the specification for CSS anchor positioning here: `https://www.w3.org/TR/css-anchor-position-1/`.

Anchor positioning is a pretty comprehensive solution to the problems of positioning one thing relative to another. It's a great addition to the language, but I still find myself having to reference the specification as there is so much to get used to.

Limitations and frustrations of anchor positioning

So far, I have extolled the virtues of CSS Anchor Positioning, but I also need to be realistic with you a moment and tell you that it can also be absolutely infuriating and fail in seemingly opaque ways at times too.

There will be occasions, once you start using anchor positioning in earnest, where the failure of an element to find its anchor, and behave as desired, can be frankly baffling.

There is a fairly detailed explanation of how an anchor is found in the *Finding an Anchor* part of the specification: `https://www.w3.org/TR/css-anchor-position-1/#target` but despite re-reading that a few times, the reasoning is not something I think is simple to understand, or to commit to memory. Sadly, as such, I don't have a succinct few simple rules to impart that cover off all eventualities for how things can go wrong.

Let me show you an example, which you can find at `example_15-15`, of a straightforward manner in which anchor positioning may not produce the result you expect. Maybe you can figure out why it doesn't work straight away?

Consider this markup.

```
<div class="parent">
    <div class="child">
        <div class="grand-child"></div>
    </div>
</div>
```

Without prior knowledge, I might want to anchor the `grand-child` element here, to the `parent` element. Let's set some very basic styles to accomplish this.

```
.parent {
    anchor-name: --parent-anchor;
    width: 800px;
    height: 400px;
    background-color: beige;
}

.child {
    position: relative;
    top: 100px;
    width: 500px;
    height: 200px;
    background-color: green;
}

.grand-child {
    width: 100px;
    height: 100px;
    background-color: gold;
    position: absolute;
    position-anchor: --parent-anchor;
    position-area: right center;
    top: 10px;
}
```

We have made the parent element the anchor element by giving it `anchor-name: --parent-anchor`.

There is then a `child` element, and within that the `grand-child` element. The `grand-child` element is absolutely positioned, and positioned 10 px from the top of the anchor, which we have set with `position-anchor: --parent-anchor;`. I would expect the positioned element to be to the right and vertically centered of the `parent` element. This is what I was expecting...

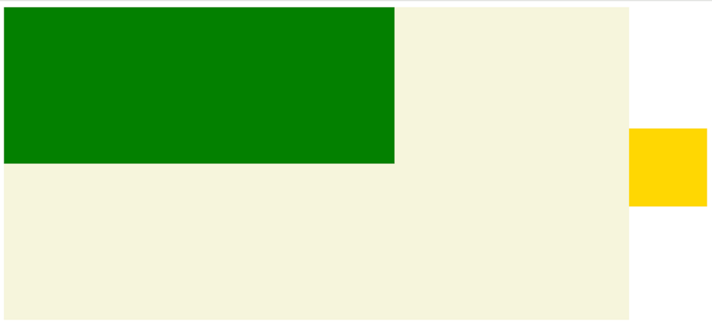


Figure 15.19: The layout I expected to see with a named anchor and anchor positioning

However, looking at this in the browser, you will see that the `grand-child` is positioning itself relative to the `child` element instead, despite us instructing it to position itself to the right center of the anchor element. How rude! This is how it is actually rendering.

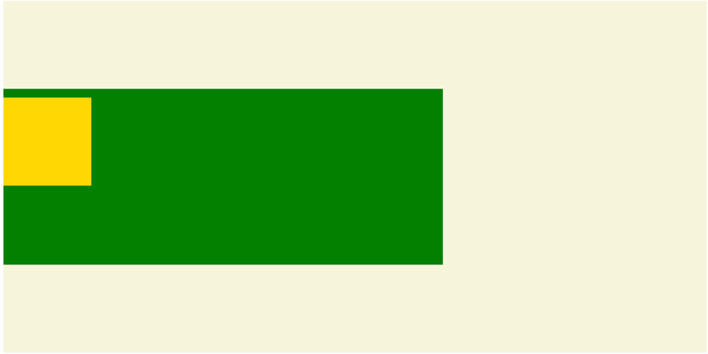


Figure 15.20: How things are actually rendering with that code

But why?

In this instance, it is the presence of the `position: relative;` on the `child` element. This is creating a containing block for the `grand-child`, and despite us declaring a `position-anchor`, it instead, as the `grand-child` is absolutely positioned, looks to its first non-static positioned `parent` element and positions itself relative to that.

There is endless nuance as to what circumstances might cause a containing block, something as seemingly unrelated to the job at hand as using a transform for instance.

The best chance to have anchor positioning work as you desire, is to not have a positioned element nested inside the element you are using as the anchor. This is by no means the only thing to consider, but it eliminates most of the common problematic scenarios I have encountered so far.

If you find yourself baffled as to why your anchor positioning is not working as intended, there is a solid guide to more common pitfalls here: `https://www.oddbird.net/2025/01/29/anchor-position-validity/`

Next, I want to look at another welcome addition to the web platform, and that's scroll-driven animations. They provide a lovely, simple way to create incredible visual effects.

Scroll-driven animations

Have you ever had a scenario where you wanted to do a certain thing with elements as they scrolled into view? Like fade or transform them in? I have, plenty of times, and in the past that meant reaching for JavaScript and `IntersectionObserver` or scroll APIs. Now, as is the trend with the things we are looking at in this chapter, we can achieve this with better results and simpler code with CSS.

> Remember, when dealing with animations, you can aid users who might not enjoy them by wrapping the effects in a suitable media query with `@media not (prefers-reduced-motion: reduce)`.

Example time! Open `example_15-08`. This is a page full of rectangles that overflow the viewport (unless you are from the future with a 30,518 px high screen, in which case it won't overflow – but how big is your desk?). Scroll the page and marvel at the beauty of those rectangles appearing in the viewport. If you can't see the example right now, know that they are being transformed in perspective, while fading in, as they enter the viewport. Here's a screenshot to illustrate.

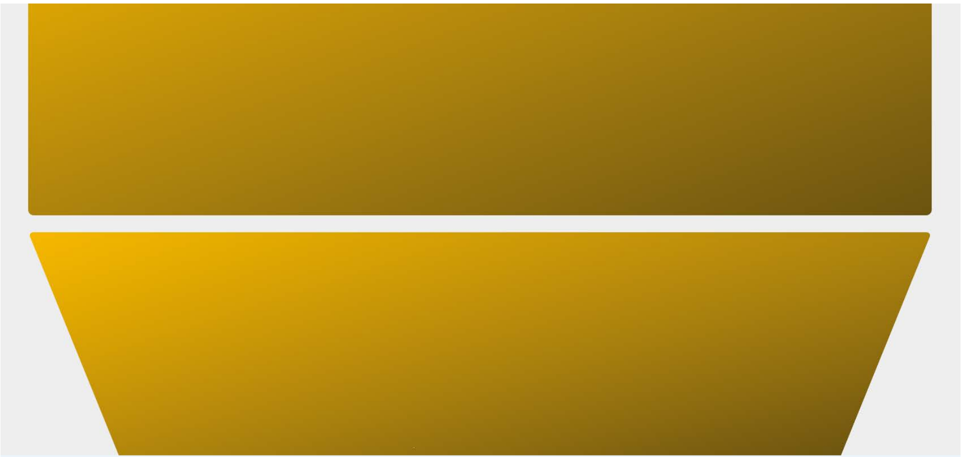


Figure 15.11: Transforming and fading in – scroll-linked animations afford a simple API for great results

The key concept to understand with scroll-driven animations is that instead of an animation being “driven” by the passing of time, the animation is “driven” by scrolling.

Here is the key code in our example. As in prior examples, I have omitted styles that are not directly related to the subject at hand:

```
.panel {
    animation: enter linear both;
    animation-timeline: view();
    animation-range: 0% 300px;
    position: relative;
    transform-origin: top center;
}

@keyframes enter {
    0% {
        transform: perspective(200px) rotateX(-90deg);
        opacity: 0;
    }
    100% {
        transform: none;
        opacity: 1;
    }
}
```

Let's get the easy stuff out of the way. At the bottom, we have some keyframes I've called `enter`, defined for an animation. We learned all about these in *Chapter 12*, *Transitions, Transformations, and Animations*. There is nothing new there; just know that this set of keyframes determines the animation that gets run. We also use the `animation` shorthand in the same way, except we don't provide a duration because remember, it is the scroll that is driving the progress of the animation, not the passage of time.

The two declarations that are new here are `animation-timeline: view()` and `animation-range: 0% 300px`. Let's consider each in turn.

The animation-timeline property

The `animation-timeline` property can be set to `scroll()` or `view()`. In our example, we have chosen `view()` because this relates to the progress of the element entering the view.

By default, `view()` is set to the block direction, so the animation will be based on the element entering vertically. However, if we were working with a horizontally scrolling container, we would set it to `view(inline)`. You can also pass `view-timeline-offset` into `view()`, to effectively make the area that is being viewed be offset by some amount. So, if I wanted the start of `view-timeline-inset` to be 50% from the bottom and stay at the top, I would do `view(0% 50%)`, and that would mean the elements would not start to appear until halfway up the viewport (you might find it useful to think about the first value as the "top" or block-start inset, and the second as the "bottom," or block-end inset). I haven't needed to use the inset values, but be aware that you can, and they must be a percentage value, or `auto` if you are resetting the inset from some other value.

Important point: the `animation-timeline` and `animation-range` properties have to be set after the `animation` shorthand declaration or they won't work. That's because `animation-timeline` and `animation-range` can't be set as part of the `animation` shorthand, but the `animation` shorthand resets all `animation-*` longhand properties, so those two properties need adding after `animation` so they do not get reset.

You may well be wondering at this point that if we can inset the timeline by passing an argument to the `view()` function, what is `animation-range` doing?

The animation-range property

The `animation-range` property is a shorthand for `animation-range-start` and `animation-range-end`, and it allows you to set how far into the view the beginning and end of the range should be. More simply, it is the area over which to execute your animation. Given we have it set in our example to `0% 300px`, what does this mean?

In this case, it means that the animation begins as soon as it enters the view, as `0%` is the beginning of the view, and the animation ends `300px` into the view. The specifications tell us that the values need to be length percentages, but px values also work fine in my experience. You can also use `calc()` if you need to produce some value that's more useful to your use case, such as `calc(100% - 150px)`.

There are keywords to consider, too, where you can subtly change behavior. The most useful ones are `entry`, `entry-crossing`, `exit`, and `exit-crossing`. I'll confess, they initially had me confused. Want to take a guess as to why you would choose `entry-crossing` over `entry`?

It really comes down to the size of the element(s) entering the view. If you choose `entry`, the animation starts as soon as a pixel of the element enters the view, and ends as soon as the last pixel of the element is in view. Pretty cool. That is until the element you decide to animate is bigger than the view. In that case, the animation would finish when the entirety of the view is covered by the element, even if some of the element were still out of view. With `entry-crossing`, the animation starts at the same point, but ends as soon as the last part of the element has crossed the "entrance" or start of the view.

There are equivalent `exit` and `exit-crossing` keywords that let you achieve the same results, but as items leave the view. The `entry`, `entry-cossing`, `exit`, and `exit-crossing` keywords are the ones I have used most frequently. But be aware that there are also `cover` and `contain`. The following is a summary of the behavior of each:

For the simplest comparison, remove the transform from the animation in the example, or toggle it off in the dev tools of the browser.

- `cover`: Means that your animation starts when the first pixel of your element is in the view and ends as the last pixel leaves
- `contain`: Starts as soon as the element is fully in view and ends as soon as the first pixel that was visible hits the end of the view
- `entry`: Starts when the element enters the view and ends when the last pixel of the element is in view
- `entry-crossing`: Starts when the element enters the view and ends when the last part of the element crosses the entry point
- `exit`: Starts when the last pixel of the element enters the view and ends when the last pixel exits

- `exit-crossing`: Starts when the first pixel of the element exits the view and ends when the last pixel exits

There is a handy visualizer and further resources at `https://scroll-driven-animations.style/tools/view-timeline/ranges/`.

Driving the animation with scroll()

I've played favorites so far and only told you about scroll-driven animations that relate to the view. That's because they are the ones I have found most useful. However, you could choose to use `scroll()` for the animation driver instead, as it is driven by the scroll position of an element, rather than the view.

The `scroll()` function can take the axis, such as `scroll(x)`, to use the horizontal scrolling as the driver; `scroll(root)`, if you want to use the document root as the container to drive from; or `scroll(nearest)` for the nearest ancestor scroll container. Finally, you can use `scroll(self)` to use the element's own box as the scroll container.

It stands to reason that if the object doing the "driving" of the animation is not a scroll container, or it is not causing scrolling, the animation will not run.

Consider `example_15-09`. We have a single 200 px square div colored pink, and the CSS looks like this:

```
body {
    min-height: 300vh;
}

.box {
    height: 200px;
    aspect-ratio: 1;
    background-color: hotpink;
    animation: twizzle linear both;
    animation-timeline: scroll();
    position: fixed;
    top: calc(50% - 100px);
}
```

```
@keyframes twizzle {
    0% {
        transform: rotate(0deg);
        left: 0;
    }
    100% {
        transform: rotate(360deg);
        left: calc(100% - 200px);
    }
}
```

The body of the document is set to be three viewports high, so as we scroll, the animation starts, and it ends when the scroll container, in this case the body, is fully scrolled. Halfway along, the box is halfway across the screen – notice the scroll bar handle in the following screenshot.

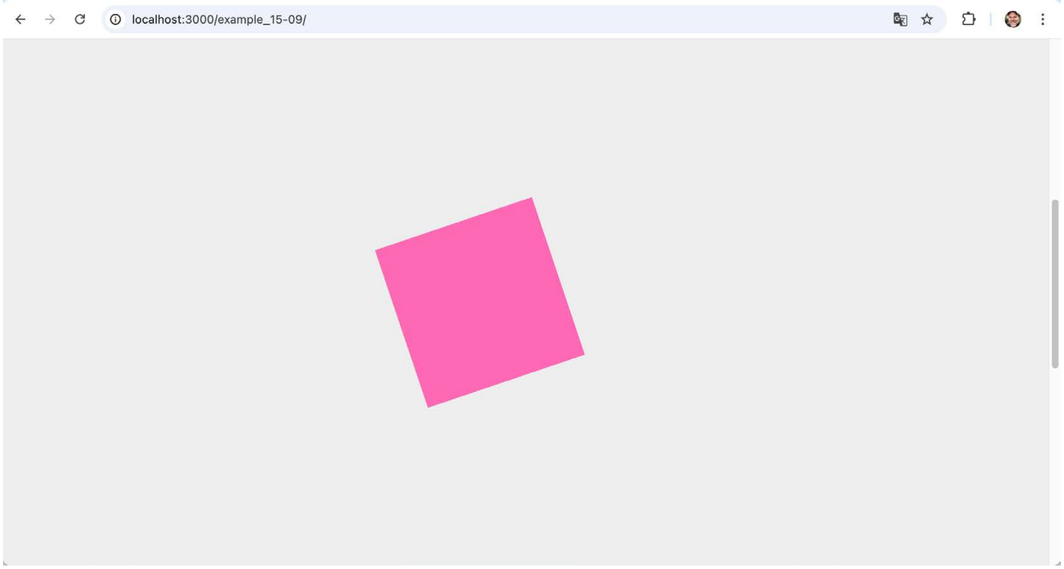


Figure 15.12: We can choose to drive animations based on the scroll container with scroll()

This isn't an entirely exhaustive explanation of all the intricacies of scroll-driven animations, but I hope it's enough to empower you to try them. Remember, conceptually, it is just substituting the passage of time driving an animation with either the position of the element relative to the viewport or the scroll position of an element. Then, with just a few extra lines of code, it's possible to make some very slick interface animations using the `@keyframes` you likely already know how to write.

I previously mentioned slick interface animations, which brings us to perhaps the most magical of new web APIs when it comes to slick visual effects, and that's view transitions.

> You can view the scroll-driven animations specification at `https://www.w3.org/TR/scroll-animations-1/` and view the latest working draft at `https://drafts.csswg.org/scroll-animations-1/`.

View transitions

The promise of view transitions is to allow a straightforward way of transitioning from one "view," which is the current visual state of an element, to another visual state, which is what will occur when you press a button, or some other action that creates a change in the DOM.

There are "same-document" view transitions that operate on elements in the same page, and "cross-document" view transitions that operate on different pages. We will look at same-document view transitions first.

It is also worth highlighting that while view transitions are supported in all the major browsers, there are additional features only in Chromium-based browsers, such as Chrome and Edge, currently.

The final caveat I would add before we get stuck in is that view transitions are not a panacea for all transitioning requirements. In some situations, it works wonderfully; in others, it can be hard work. Hopefully, by the end of this section, you will have a solid feel for the kinds of scenarios to apply them in.

Same-document view transitions

You need a little JavaScript to start a view transition, because there has to be a change in the DOM, and you need to tell the browser that when that change happens, you want to transition the view.

We are going to look at an example first of a scenario that really suits view transitions, and then move on from that to consider scenarios that will expose its limitations.

Start with `example_15-10`. We have a crude representation of 52 playing cards. Clicking the **Deal!** button at the bottom right adds a class of "dealt" to the top card in the "pack," which is the last in the DOM. In CSS, when a card gets the `.dealt` class added, its visual position is changed from the bottom to the top of the screen, and depending upon whether it is first, second, third, or fourth in terms of `nth-child`, it gets positioned along the top. After clicking the **Deal!** button four times, you should see something like this:

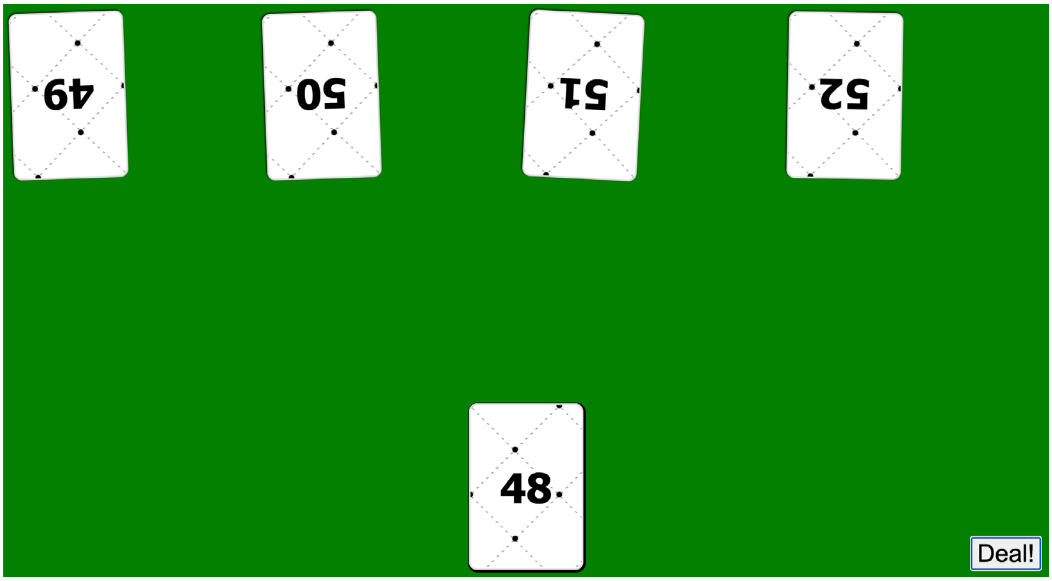


Figure 15.13: Without view transitions, the cards snap into place

Right now, the cards are just snapping from one position to another. There is no view transition happening. The JavaScript to make the interactions happen as they are at the minute looks like this:

```
const btn = document.querySelector("#deal");
let dealt = 0;

btn.addEventListener("click", () => {
    const cards =
        Array.from(document.querySelectorAll(".card:not(.dealt)"));
    let nextCard = cards[cards.length - 1];
    nextCard.classList.add("dealt");
    dealt++;
    nextCard.style.zIndex = dealt;
});
```

When the button is clicked, a variable increments a number assigned to `dealt`. We then select all the cards, and the last card that does not have `.dealt` class on it gets `.dealt` added. The `z-index` is added at the same time, assigned to the ever-incrementing `dealt` number to ensure the last-dealt card appears on top of any prior ones.

To add a default view transition, all we need to do is wrap the adding of the class in a `document.startViewTransition()` method. So, our amended JavaScript would look like this:

```
document.startViewTransition(() => {
    nextCard.classList.add("dealt");
});
```

If you refresh the page after adding that, you will get a simple cross-fade animation as you click **Deal!**. Cross-fade is the default transition.

So, what is actually going on under the hood?

Changing default durations

When a view transition is started, an image is effectively made of the old state and the new one. These two can be selected in CSS with pseudo-selectors. For example, to extend the default cross-fade duration, you could add this:

```
::view-transition-old(root),
::view-transition-new(root) {
    animation-duration: 5s;
}
```

If you refresh the page and click again, you will see the cross-fade now takes five seconds, which is quite slow and boring, but at least you know how to do that now.

A cross-fade is better than nothing, but visually, what I really want is to see the cards flying across the table when the **Deal!** button is clicked, and they move to their new position. But that isn't happening. That's because of a very important point.

Every element to transition needs a unique view-transition-name value

Read this next sentence a few times, because this is the reason for most frustration when you first start using view transitions. *Every* element that you want to take place in a view transition needs its own `view-transition-name`. To be clear, that is every element. Not every class of element – every element itself. In this example, if we have 52 cards, we need 52 `view-transition-name` values!

But don't despair! We have a couple of choices, and by the time you read this in the future, it might not even be something to worry about. First, let's look at the easy but more limited option. There is a newer value that can be used for `view-transition-name`, and that is `match-element`. We can add this to our `.card` class, and this magically creates a distinct `view-transition-name` in the background for each instance. Amazing!

A small catch: as of this book being published in late 2025, it is not supported in Safari. Whomp whomp.

If you find yourself in that scenario, my advice is to head back into the JavaScript and add a `view-transition-name` dynamically. You can do that with a loop across all the relevant elements in one go, or add it as it is clicked, as we will do next:

On each click of the button, we set the `view-transition-name` and use the count of clicks to increment the name so it ends up being `name-1`, `name-2`, and so on. Here is the JavaScript that runs each time the button is clicked:

```
btn.addEventListener("click", () => {
    const cards =
        Array.from(document.querySelectorAll(".card:not(.dealt)"));
    let nextCard = cards[cards.length - 1];
    dealt++;
    nextCard.style.zIndex = dealt;
    nextCard.style.viewTransitionName = `name-${dealt}`;
    document.startViewTransition(() => {
        nextCard.classList.add("dealt");
    });
});
```

With an individual `view-transition-name` added, head back to the browser and refresh, and enjoy seeing those cards flying across the table on each click!

You know how to amend the duration of the animation too, so play around with that to get your preferred dealing speed…

At least you might assume you can do that, but because we needed an individual `view-transition-name` for each element, we can’t just select them all in one go in CSS. Despite `view-transition-group` being able to take the universal selector, you might try to do this:

```
::view-transition-group(*)) {
    animation-duration: 5s;
}
```

And sadly, each transition will take the five seconds, but as each card, by necessity, has its own `view-transition-name`, you don’t actually see the transition of the card moving. The card moves into place but does not animate there, but it is still five seconds before the next card can do anything. Very frustrating.

It is possible to debug animations and view transitions in Chrome with the dedicated animations tab in **Developer Tools**: `https://developer.chrome.com/docs/devtools/css/animations#view-transition`.

Selecting simlilar transitions with view-select-class

Initially, it was necessary to add a `view-transition-group` for every `view-transition-name` for which we wanted to change the animation duration, but thankfully, there is a newer property that allows us to work around this problem more easily. We can add a `view-transition-class` to each element that we want to share the same `view-transition-group` styles. In our `.card` selector, we can add the following declaration:

```
view-transition-class: card;
```

Here, `card` is any name you choose. I chose `card` because it is for all cards, but it could be `banana` if you would rather. Notice that there is no period before the name here, but there *is* when you reference it in the `view-transition-group` like this:

```
::view-transition-group(.card) {
    animation-duration: 1s;
}
```

And now, each card flies into position over one second.

You can view the “finished” version of this example in `example_15-11`.

You can change the animations as you would any other by assigning keyframes and different timing functions to the `::view-transition-old` and `::view-transition-new` pseudo-selectors.

Pitfalls with view transitions

View transitions are great and allow us to produce some incredible animations and transitions between states, but they can also be the source of great frustration. Consider `example_15-12`. It's similar to the prior example but with more text in each card, and each card has a different aspect ratio in its end state.

As you click **Deal!**, you can see an odd trail of the text.

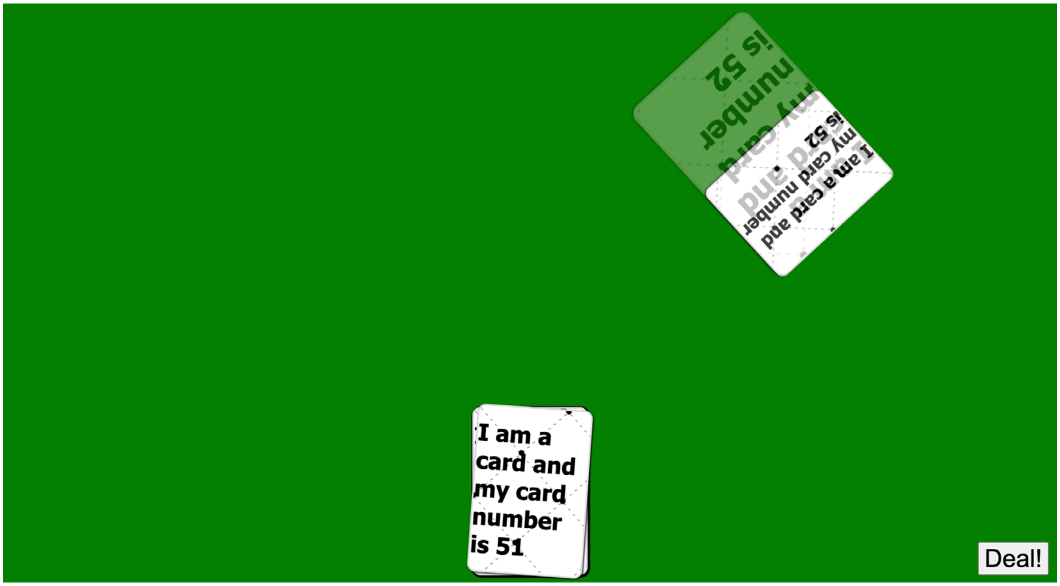
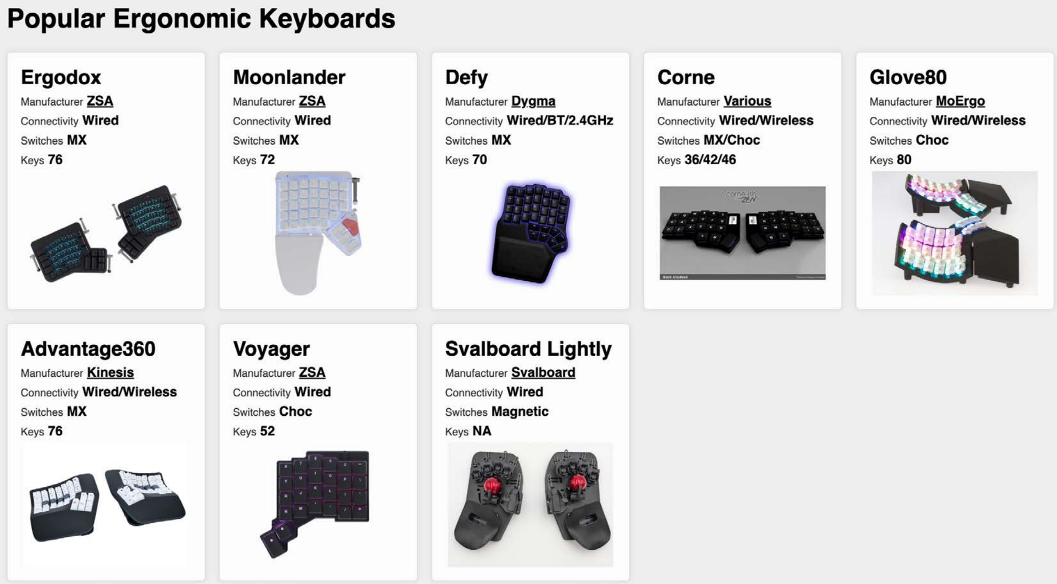

I am a card and my card number is 52
I am a card and my card number is 51
Deal!

Figure 15.14: View transitions struggle with aspect ratio changes by default

The problem occurs because the shape/aspect ratio of the text changes from one state to the other, and the view transition is not able to sensibly transition between them. If this is a situation you find yourself in, I would encourage you to read `https://jakearchibald.com/2024/view-transitions-handling-aspect-ratio-changes/` for some good tips and explanations of the finer details.

You may also struggle to get a perfect transition when something is `display: none` in one state and shown in the next. The view transition has nothing to move from, and so the newly displayed item can seemingly just appear. Consider `example_15-13`, which is more involved than the previous examples. At the outset, it looks like this.



Popular Ergonomic Keyboards

Ergodox
Manufacturer **ZSA**
Connectivity **Wired**
Switches **MX**
Keys **76**

Moonlander
Manufacturer **ZSA**
Connectivity **Wired**
Switches **MX**
Keys **72**

Defy
Manufacturer **Dygma**
Connectivity **Wired/BT/2.4GHz**
Switches **MX**
Keys **70**

Corne
Manufacturer **Various**
Connectivity **Wired/Wireless**
Switches **MX/Choc**
Keys **36/42/46**

Glove80
Manufacturer **MoErgo**
Connectivity **Wired/Wireless**
Switches **Choc**
Keys **80**

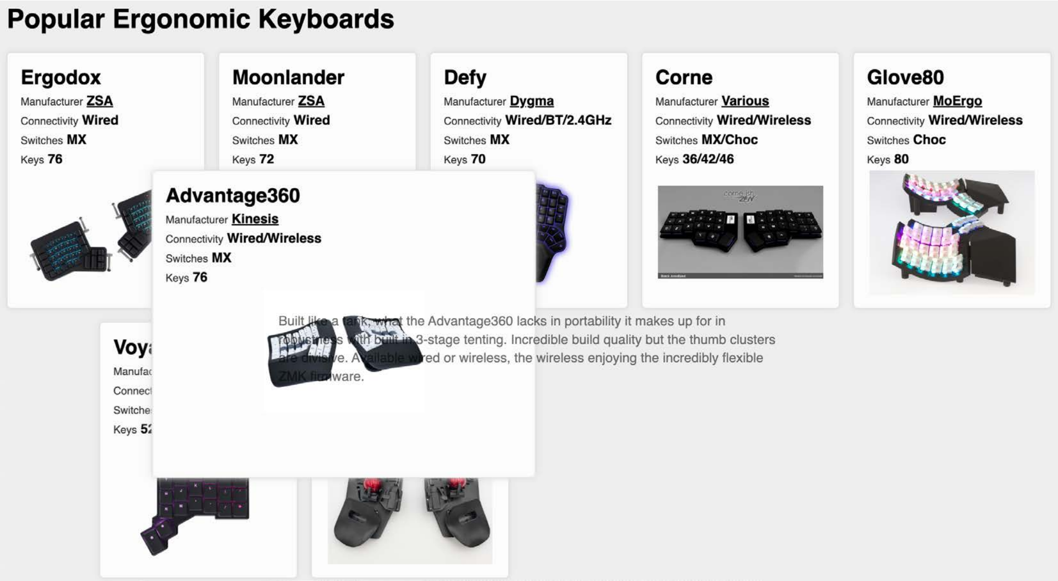
Advantage360
Manufacturer **Kinesis**
Connectivity **Wired/Wireless**
Switches **MX**
Keys **76**

Voyager
Manufacturer **ZSA**
Connectivity **Wired**
Switches **Choc**
Keys **52**

Svalboard Lightly
Manufacturer **Svalboard**
Connectivity **Wired**
Switches **Magnetic**
Keys **NA**

Figure 15.15: A more complicated example with changing aspect ratios and elements hidden at the outset

But when you click on one of the boxes, with a slow animation time, it is easy to see some issues. While the aspect ratio changes of the other text have been dealt with, the paragraph of information that was `display: none` and is now being shown appears floating instead of part of its container.



Popular Ergonomic Keyboards

Ergodox
Manufacturer **ZSA**
Connectivity **Wired**
Switches **MX**
Keys **76**

Moonlander
Manufacturer **ZSA**
Connectivity **Wired**
Switches **MX**
Keys **72**

Defy
Manufacturer **Dygma**
Connectivity **Wired/BT/2.4GHz**
Switches **MX**
Keys **70**

Corne
Manufacturer **Various**
Connectivity **Wired/Wireless**
Switches **MX/Choc**
Keys **36/42/46**

Glove80
Manufacturer **MoErgo**
Connectivity **Wired/Wireless**
Switches **Choc**
Keys **80**

Advantage360
Manufacturer **Kinesis**
Connectivity **Wired/Wireless**
Switches **MX**
Keys **76**

Built like a tank, what the Advantage360 lacks in portability it makes up for in robustness with built in 3-stage tenting. Incredible build quality but the thumb clusters are divisive. Available wired or wireless, the wireless enjoying the incredibly flexible ZMK firmware.

Voy
Manufac
Connect
Switche
Keys **52**

Figure 15.16: With a slower animation duration, the shortcomings are exposed

Sometimes, there is a lot of work to be done in getting view transitions to behave exactly as you might like. Other times, you can cheat the effect by speeding up the animation. I think `example_15-03` is passable if you speed the transition up to only `.25s` at the top of the CSS. Give that a try and see what you think!

For some situations, view transitions can feel like magic; for others, they can seem like the work of the devil!

Cross-document view transitions

As well as being able to perform view transitions on different "states" of the same page, you can also make use of the closely related cross-document view transitions. As long as the pages exist on the same origin, you can transition between them.

Same origin means that you could transition between pages at `https://mysite.com/1` and `https://mysite.com/2` but you could not transition between pages at `https://mysite.com/1` and `https://cakes.mysite.com/2` as the subdomain is a different origin.

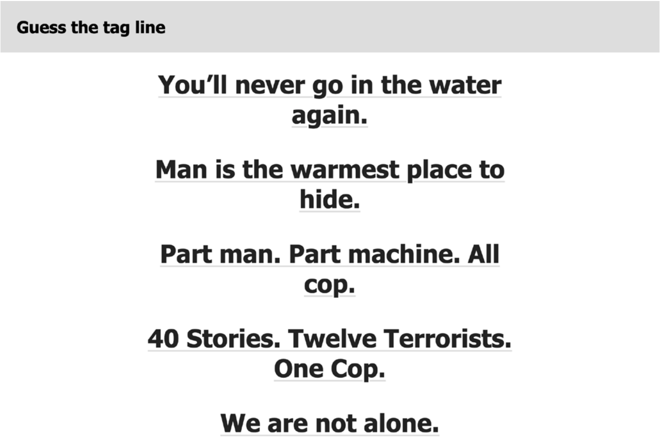
To opt in to the default view transition, we need to add the following to the applicable pages. I typically just add this somewhere near the top of the shared stylesheet:

```
@view-transition {
  navigation: auto;
}
```

The `@view-transition` at-rule is all you need to add. Unlike same-document view transitions, there is no JavaScript needed. The only kinds of navigation between pages that would not now be "transitionable" would be if the user entered another page at the same origin using the address bar, or refreshed the page. In those cases, the page would load directly. After a browser-specific amount of time, the view transition would also error, but it would not stop the page from appearing normally if it subsequently loaded.

To see the view transition across the pages, open `example_15-14`. You will need to run this example as a server, so if you have npm installed, browse to the relevant example folder and run `npx serve` (or use any other local server). That will spin up a server at `localhost:3000`, which works for cross-document view transitions as pages will have the same origin.

The index page looks like this.



Guess the tag line

You'll never go in the water again.

Man is the warmest place to hide.

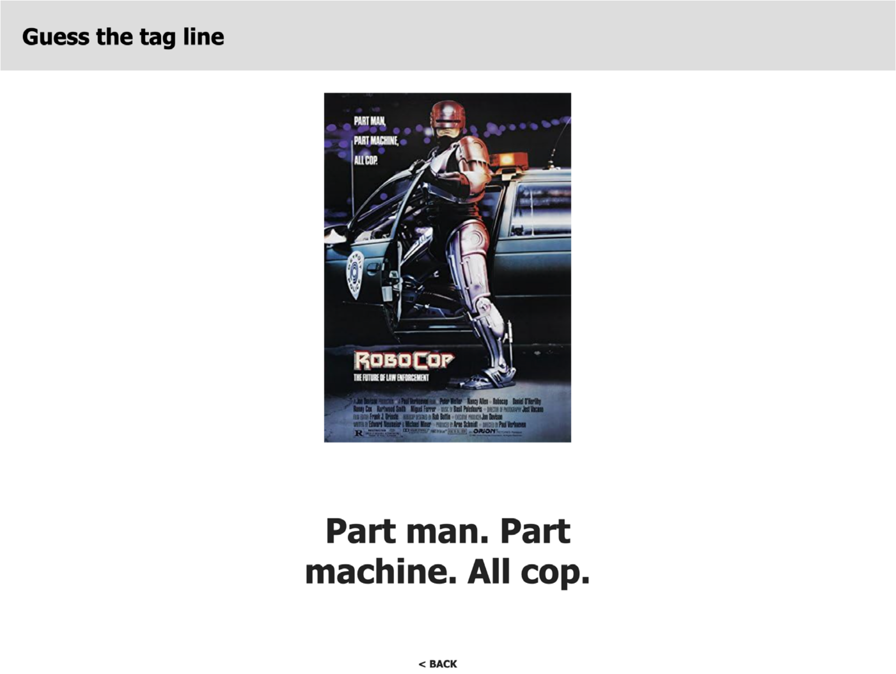
Part man. Part machine. All cop.

40 Stories. Twelve Terrorists. One Cop.

We are not alone.

Figure 15.17: Cross-document view transitions don't require any JavaScript to set up

If you click the movie taglines, you will navigate to a page showing the appropriate movie poster. But notice how the gray header stays in place and the content fades in as you move between pages? For such little work, it's a surprisingly nice effect.



Guess the tag line

Part man. Part machine. All cop.

< BACK

Figure 15.18: With just a few lines of CSS, you get a pleasant cross-fade across pages

Despite the simplistic approach we have taken in this example, it is possible to go much further with cross-document view transitions, using much the same mechanics as we looked at for same-document view transitions. View transitions are also under active development in browsers, so it is worth keeping an eye on the developing level 2 version of the specification at `https://drafts.csswg.org/css-view-transitions-2/`.

Summary

Well done for making it through another packed chapter. I hope you learned as much reading that as I did putting it together! Popovers, anchor positioning, scroll-driven animations, and view transitions – these are incredible new features in browsers now, with support only getting better. As I have said before, try not to worry about the syntax, but try and understand the use cases and capabilities these features offer. If you are a fan of AI tools, it's much easier to get the latest model to do your bidding if you have a solid grasp on the fundamentals of the feature, even if you might not recall every piece of related syntax.

CSS has moved at an incredible pace in the last few years, and it seems that never a week goes by without me coming across some new functionality I have missed or that is coming to browsers soon. It's certainly easy to feel overwhelmed with the pace of changes in frontend development. However, you shouldn't. With some common sense and a basic interest in what's going on, I feel it is easy enough to stay in the game.

The more philosophical musings on how to deal with modern frontend development, and many of the other new things coming to CSS and HTML, could be a chapter all of its own.

Which is handy, because that's exactly what our last chapter is all about.

You've made it this far; walk this last mile with me, will you?

16

Bonus Techniques and Parting Advice

In my favorite stories and films, there's often a scene where a mentor passes on valuable advice and some magical items to the hero. You know those items will prove useful; you just don't know when or how.

Well, I'd like to assume the role of mentor in this final chapter—besides, my hair has waned, and I don't have the looks for the hero role. I would like you, my fine apprentice, to spare me just a few more moments of your time while I offer up some final words of advice before you set forth on your responsive quest.

This chapter will be half philosophical musings and guidance, and half a grab-bag of unrelated tips and techniques. I hope, at some point in your responsive adventures, these tips will prove useful.

Here's the grab-bag of tips we'll cover:

- Truncating text: single-line and multi-line
- Creating horizontal scrolling panels
- Customizing scrollbars
- Using CSS Scroll Snap
- Smooth scrolling with the CSS `scroll-behavior` property

And here are the topics I will suggest guidance on:

- Getting designs in the browser
- Testing on real devices
- Embracing progressive enhancement
- Defining a browser support matrix
- Avoiding CSS frameworks in production
- Hiding, showing, and loading content at different viewports
- Validators and linting tools
- Performance
- The next big things

Right, now pay attention, 007...

In my day-to-day work, I've found that I use some CSS features constantly and others hardly ever. I thought it might be useful to share those techniques I've used most often, especially those that pop up time and again with responsive projects.

These are in no particular order. None is more important than any other. Just dip in if and when you need them.

Truncating text

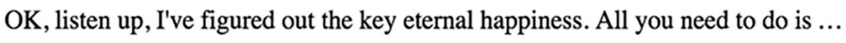
Sometimes, you encounter a situation where space is limited and you would prefer to see text truncated rather than wrapped. We've been trained to spot truncated text on websites for years with the ellipsis symbol, "...".

Single-line truncation

This is straightforward in CSS. Consider this markup (you can view this example in `example_16-01`):

```html
<p class="truncate">
  OK, listen up, I've figured out the key to eternal happiness.
  All you need to do is eat lots of scones.
</p>
```

However, we actually want to truncate the text at 530px wide, so it looks like this:



OK, listen up, I've figured out the key eternal happiness. All you need to do is ...

Figure 16.1: Truncation is handy when keeping dimensions constant is of paramount importance

Here is the CSS to make that happen:

```
.truncate {
  width: 530px;
  overflow: hidden;
  text-overflow: ellipsis;
  white-space: nowrap;
}
```

Each one of those properties is needed to make the truncation occur.

You can read the specification for the `text-overflow` property here: `https://drafts.csswg.org/css-overflow-3/#text-overflow`.

Whenever the width of the content exceeds the `width` defined, it will be truncated. The `width` can just as happily be set as a percentage, such as 100% if it's inside a flexible container.

We set `overflow: hidden` to ensure that anything that overruns the box is hidden.

When the content does overrun, `text-overflow: ellipsis` creates the ellipsis symbol in the correct place to indicate the overrunning content. This could be set to `clip` if preferred. In that instance, the content just gets clipped where it overflows, possibly mid-character.

The `white-space: nowrap` property/value pair is needed to ensure that the content doesn't wrap inside the surrounding element, which is what it would do by default.

Very occasionally, I've encountered an instance where, despite these properties being present, the text is not being truncated. In every case, this has been when the box has been in one or more flex containers, and the solution is to go up the DOM tree, ensuring that any ancestor flex containers also have `overflow: hidden` applied.

Multi-line truncation

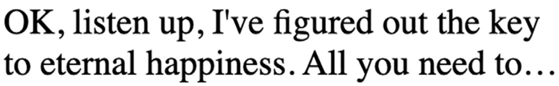
There still isn't a perfect way to do multi-line truncation, despite there being a specification that seems to make perfect sense: `https://drafts.csswg.org/css-overflow-4/#line-clamp`.

What follows is based on an older, proprietary WebKit way of doing things that has been around for so long now that browser providers have actually implemented it in their own browsers, with a `-webkit-` prefix and all!

Right now, you can use `-webkit-line-clamp` like this, which you can view in `example_16-02`:

```
.multi-line-truncate {
    width: 250px;
    text-overflow: ellipsis;
    overflow: hidden;
    -webkit-line-clamp: 2;
    -webkit-box-orient: vertical;
    display: -webkit-box;
}
```

That produces an effect like this:



OK, listen up, I've figured out the key
to eternal happiness. All you need to…

Figure 16.2: Multi-line truncation is possible with caveats

This seems like exactly what we want. However, be advised that the implementation is brittle. Here are some reasons why. Firstly, if you happen to omit the `overflow: hidden` declaration, you still get the ellipsis at the line length you specified, but the text continues. Worse, even with those declarations all added, if you happen to add padding inside the box, you get a similar effect of an ellipsis at the specified line, but with additional text still visible.

It can certainly help you out in a pinch, but be aware of the caveats.

Creating horizontal scrolling panels

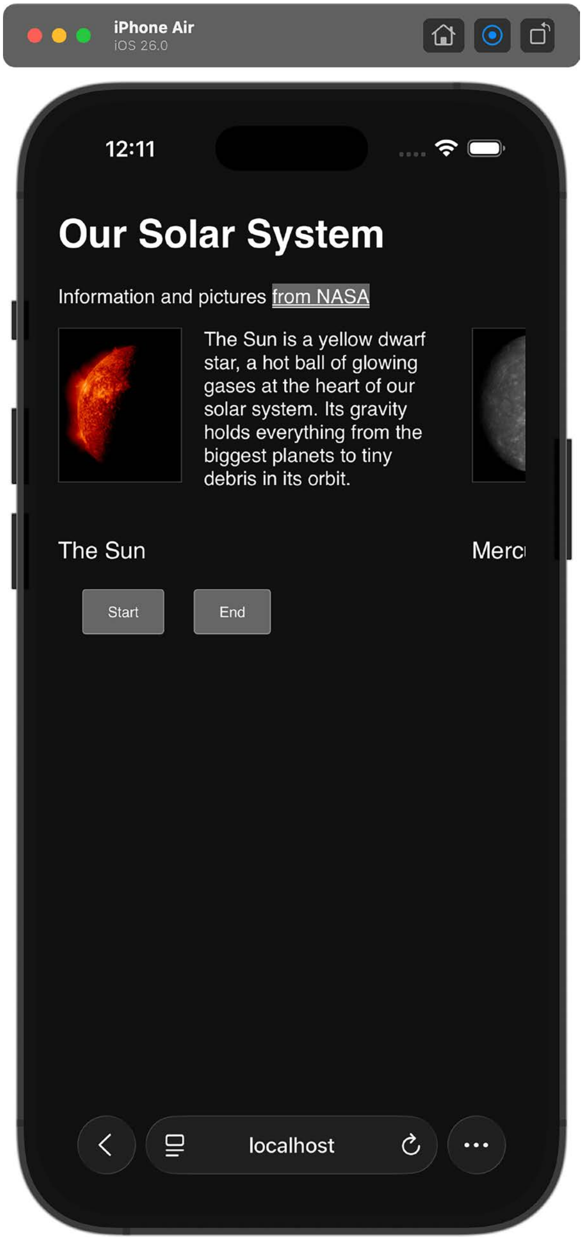
When I say *horizontal scrolling panel*, hopefully you know the kind of thing I mean.

Horizontal scrolling panels are common on the iOS App Store and Google Play Store for showing panels of related content (movies, albums, and more). Where there is enough horizontal space, all of the items are viewable. However, when space is limited (think mobile devices), the panel is scrollable from side to side.

We made a horizontal scrolling panel together in `example_05-10` in *Chapter 5*, *Layout with CSS Grid*. That was the basis for what ended up as `example_16-03`, which we are going to walk through in the next sections. If you open `example_16-03` in the browser, you can see all the extra goodness applied.

we had was already perfectly functional, but there are some newer, scrolling-related treats we can add with CSS that really elevate the experience of using it.

Our horizontal slider looks like this on an iPhone Air:



Our Solar System

Information and pictures from NASA

The Sun is a yellow dwarf star, a hot ball of glowing gases at the heart of our solar system. Its gravity holds everything from the biggest planets to tiny debris in its orbit.

iPhone Air
iOS 26.0
12:11
The Sun
Merc
Start
End
localhost

Figure 16.3: A horizontal scrolling panel

The markup pattern looks like this; note that I'm just showing the first item in the list for brevity:

```
<div class="Scroll_Wrapper">
    <figure id="start" class="Item">
        <img src="Sun.jpg" alt="Our Sun in space" />
        <figcaption class="Caption">The Sun</figcaption>
        <p>
            The Sun ...
        </p>
    </figure>
```

To get the scrolling working, we just need a wrapper that is narrower than the sum of its contents and to set it to overflow automatically in the x axis. That way, it won't scroll if there is enough space, but it will if there isn't.

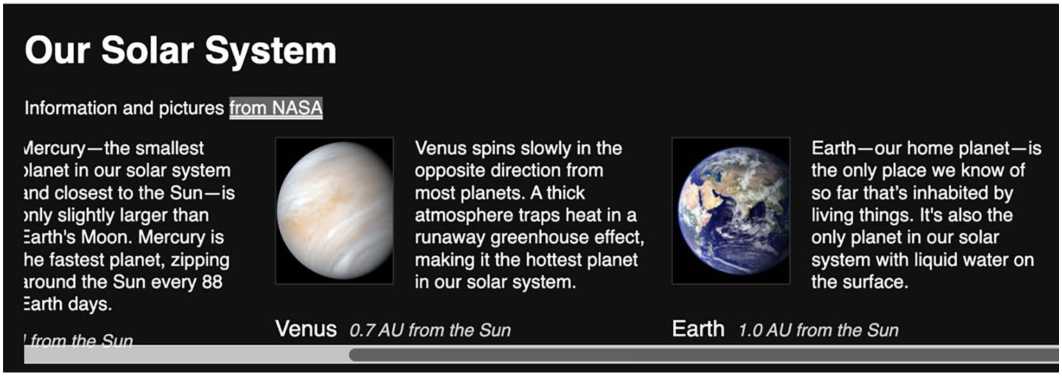
With Grid, the main scroller is set like this:

```
.Scroll_Wrapper {
    display: grid;
    width: 100%;
    overflow-y: hidden;
    overflow-x: auto;
    gap: 0 20px;
    grid-template-rows: auto auto;
    grid-auto-columns: auto;
    grid-auto-flow: column;
}
```

No matter how many items are added to the grid, it will continue to add them as columns. Skip back to *Chapter 5*, *Layout with CSS Grid*, if you need a refresher on any of the properties and values used.

One thing you might not be very happy about is the default scrollbar. This is typically quite elegant on touch devices but a little more unrefined on desktops.

Here it is in Chrome:



Our Solar System

Information and pictures from NASA

lercury—the smallest
lanet in our solar system
nd closest to the Sun—is
nly slightly larger than
arth's Moon. Mercury is
he fastest planet, zipping
round the Sun every 88
arth days.

from the Sun

Venus spins slowly in the opposite direction from most planets. A thick atmosphere traps heat in a runaway greenhouse effect, making it the hottest planet in our solar system.

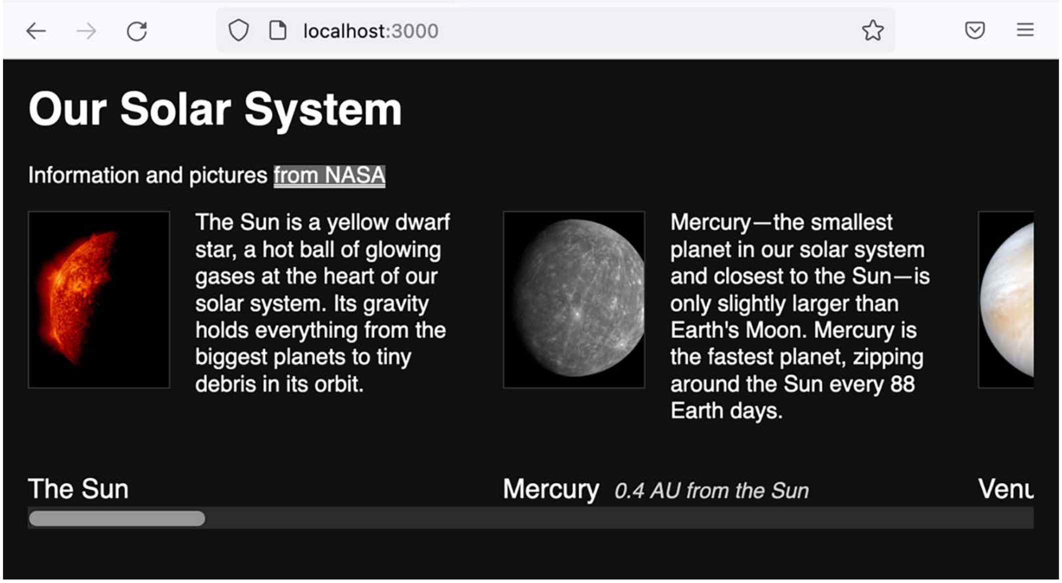
Venus *0.7 AU from the Sun*

Earth—our home planet—is the only place we know of so far that's inhabited by living things. It's also the only planet in our solar system with liquid water on the surface.

Earth *1.0 AU from the Sun*

Figure 16.4: Default scrollbars in Chrome

And here it is in Firefox (both browsers on macOS):



localhost:3000

Our Solar System

Information and pictures from NASA

The Sun is a yellow dwarf star, a hot ball of glowing gases at the heart of our solar system. Its gravity holds everything from the biggest planets to tiny debris in its orbit.

The Sun

Mercury—the smallest planet in our solar system and closest to the Sun—is only slightly larger than Earth's Moon. Mercury is the fastest planet, zipping around the Sun every 88 Earth days.

Mercury *0.4 AU from the Sun*

Venu

Figure 16.5: Default scrollbar on Firefox (macOS)

I actually think the default appearance on Firefox is very nice, and I'm not sure I would do much to change it. However, the Chrome one could maybe do with a little work.

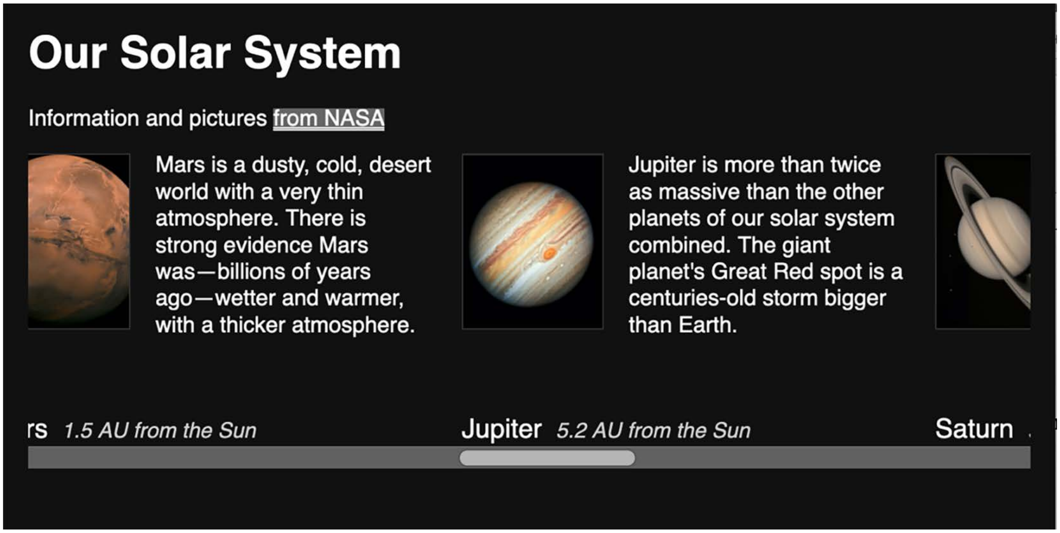
Styling scrollbars

Nowadays, thanks to standards and modern CSS, we can make visual changes to the way scrollbars look.

Let's start with the specification and see what that gives us. It's important to note that the specification is purposefully more limiting than some of the non-standard scrollbar controls. This is because it is not possible to know exactly what constituent parts a scrollbar will have on any given system. As such, the specification limits us to the color of the scroll track and **thumb** (that's the term for the part you click/touch and drag), and the width of the scrollbar itself. Here it is applied:

```
.Scroll_Wrapper {
    display: grid;
    width: 100%;
    overflow-y: hidden;
    overflow-x: scroll;
    gap: 0 20px;
    grid-template-rows: auto auto;
    grid-auto-columns: auto;
    grid-auto-flow: column;
    /* standards way */
    scrollbar-color: #b5b5b5 #616161;
    scrollbar-width: thin;
}
```

With `scrollbar-color`, you give it two values; the first is the color for the scrollbar "thumb," and the second is for the track of the scrollbar. For `scrollbar-width`, you have only three choices: `auto`, `thin`, or none. I've opted for `thin`, but as you can see in my case, in Firefox, that has made very little difference:



Our Solar System

Information and pictures from NASA

Mars is a dusty, cold, desert world with a very thin atmosphere. There is strong evidence Mars was—billions of years ago—wetter and warmer, with a thicker atmosphere.

Jupiter is more than twice as massive than the other planets of our solar system combined. The giant planet's Great Red spot is a centuries-old storm bigger than Earth.

Figure 16.6: scrollbar-width has a vague effect in Firefox (macOS)

Read the *Scrollbars Styling* specification at `https://www.w3.org/TR/css-scrollbars-1/`.

Scrollbar color accessibility

However, before we get into the non-standard, proprietary way that Safari and older Chromium-based browsers support, here's an important point. If you are going down the route of altering scrollbars in any significant manner, ensure that you consider accessibility. In the preceding example, I went for a 3:1 color contrast ratio of track to background, and thumb to track. That should be the minimum you opt for, despite the fact that, often, the host operating system's own scrollbars do not adhere to those contrast ratios!

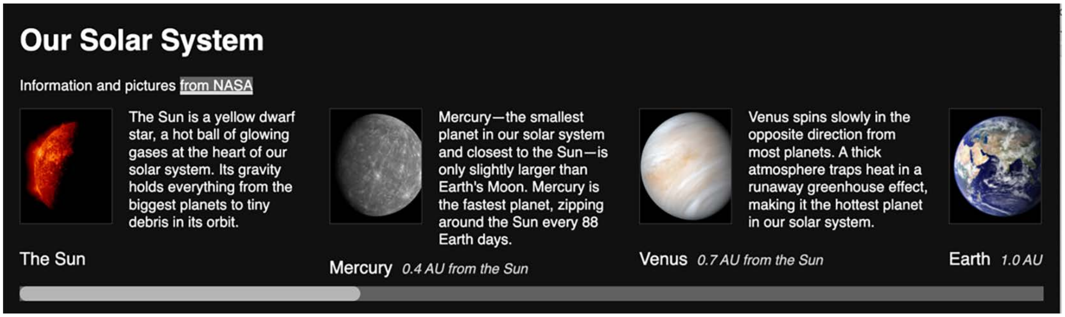
If you want a deeper dive into the accessibility considerations when styling scrollbars, I recommend the following: `https://adrianroselli.com/2019/01/baseline-rules-for-scrollbar-usability.html`.

Proprietary -webkit- scrollbar styling

For WebKit and Chromium-based browsers such as Chrome and Edge, if you don't add the more recent, specification-based styling, you can still make use of the older proprietary WebKit-prefixed styling. This offers far more permutations but is not supported across all browsers, Firefox being a notable exception. Here is an example of using the various WebKit pseudo-elements:

```css
.Scroll_Wrapper::-webkit-scrollbar {
    width: 25px;
    background-color: #616161;
}

.Scroll_Wrapper::-webkit-scrollbar-thumb {
    background-color: #b5b5b5;
    border-radius: 12px;
}
```

And in Chrome, this is the effect that it has:



Our Solar System

Information and pictures from NASA

The Sun is a yellow dwarf star, a hot ball of glowing gases at the heart of our solar system. Its gravity holds everything from the biggest planets to tiny debris in its orbit.

The Sun

Mercury—the smallest planet in our solar system and closest to the Sun—is only slightly larger than Earth's Moon. Mercury is the fastest planet, zipping around the Sun every 88 Earth days.

Mercury *0.4 AU from the Sun*

Venus spins slowly in the opposite direction from most planets. A thick atmosphere traps heat in a runaway greenhouse effect, making it the hottest planet in our solar system.

Venus *0.7 AU from the Sun*

Earth *1.0 AU*

Figure 16.7: Using non-standard scrollbar styling in Chrome

The scrollbar also looks nearly identical in desktop Safari, but Firefox ignores it entirely.

There are other `-webkit-` prefixed scrollbar properties; the aim here is merely to get us some equivalence to the "official" declarations we set. Remember, regardless of what you set, mobile browsers tend to do their own thing anyway.

I'm going to revert back to the standards approach, and leave the `-webkit-` version commented out; the standards provide something acceptable and fairly consistent across browsers.

Now, I want to show you another lovely little scrolling-based extra that is simple to add but produces a very polished result and satisfying experience for your users.

CSS Scroll Snap

CSS Scroll Snap snaps the scrolling of content to predefined points in the container. Again, it is a user interface pattern that is commonplace in the interfaces of native applications, app stores, and things like carousels. However, historically, such functionality required JavaScript to implement to get anything approaching a comparable experience.

> You can read the official specification for CSS Scroll Snap at `https://www.w3.org/TR/css-scroll-snap-1/`.

Let's use CSS Scroll Snap to add scroll snap functionality to our horizontal "Solar System" container.

The scroll-snap-type property

First of all, we define the `scroll-snap-type` property for our scrolling container. This is where we can decide whether we want the container to scroll snap to items in the x, y, or both axes.

This property also allows us to define the *strictness* of the scroll snapping that is applied. The `mandatory` value is the one I always opt for unless I'm resetting scroll snapping, at which point you can use `none`, which will revert scroll snap to behave like a standard scroll container.

What `mandatory` does is ensure that if an item is partially scrolled within the container (for example, the left-hand side is off the viewport by a certain amount), then it will be "snapped" into or out of view.

There is also a `proximity` value. What this does is leave the snapping to the discretion of the browser.

It is important to understand why you may occasionally want and need to use `proximity`. Suppose you had items in a carousel that were always bigger than the viewport. Because it would always snap with `mandatory`, you may have trouble advancing the scroll panel. Setting it to `proximity` means that content will remain accessible as the browser will determine how much room should be left to accommodate scrolling.

Despite that, in our instance, since we know that the width of our cells won't exceed the width of the viewport, we will apply `mandatory` for a consistent experience.

So, our container will be amended with this extra line:

```
.Scroll_Wrapper {
    /* other styles */
    scroll-snap-type: x mandatory;
}
```

Now, if you refresh the browser and try the scroll panel, you will likely be disappointed. That, by itself, doesn't do anything. That's because we also need to apply the `scroll-snap-align` property to the child items.

The scroll-snap-align property

The `scroll-snap-align` property controls where an item inside a `scroll-snap` container will snap to. The options are as follows:

- none to reset an item
- `start` to snap the item to the beginning of a scroll snap area
- end to snap the item to the end of a scroll snap area
- `center` to snap the item to the center of the scroll snap area

So, let's add this property to `.Item` and set the value to make it snap to the start:

```
.Item {
    /* other styles */
    scroll-snap-align: start;
}
```

Now, if you scroll, you'll see that the items snap within their container.

The scroll-snap-stop property

By default, an excessive scroll action, such as whisking your finger across the container on mobile, can send several items past the scroll snap point. That might not be something you want to allow. With `scroll-snap-stop`, it is possible to make the browser stop at each snap point with each scroll action. There are just two values to choose from:

- `normal`, which is what the browser does by default
- `always`, which ensures that no scroll snap points are skipped

You add this declaration to the items inside the scroll container.

CSS Scroll Snap is a hugely satisfying piece of functionality to use! In just a few lines, we are able to accomplish in CSS what used to take whole libraries of JavaScript to achieve. All this talk of scrolling brings me to another simple but effective feature I need to relate.

Smooth scrolling with CSS scroll-behavior

One of the oldest capabilities of HTML is the ability to anchor to different points in a document. You set a link on the page, and rather than it sending the user to another web page, the user is instead instantly taken to a different point on the same page.

Typically, such functionality is on the *y* axis, or down a page. However, it works just as well horizontally in the *x* axis.

Historically, jumping to an anchor link has always been a little jarring. As the user is instantly transported to the new point on the page, there is no affordance to communicate to the user what just occurred. People have solved this common issue over the years with the help of JavaScript by animating the scroll action.

Now the web platform has provided us with a simple API to achieve this effect natively, which can be used by either JavaScript or CSS. For CSS, it is the `scroll-behavior` property.

I'm going to add `"start"` and `"end"` anchor points to each end of the scroll panel we just made and add two links below the scrolling panel.

So, on the first scrolling item, we add `id="start"`:

```
<figure id="start" class="Item">
```

And on the last, we add this:

```
<figure id="end" class="Item">
```

And our two links look like this:

```
<a class="ScrollBtn" href="#start">Start</a>
<a class="ScrollBtn" href="#end">End</a>
```

Now, our mechanism is set up as traditional anchor points. As you might imagine, by default, clicking on **End** instantly scrolls the panel to the end. However, if we add `scroll-behavior: smooth` to our scroll panel, we get a silky-smooth scroll behavior instead:

```
.Scroll_Wrapper {
    display: grid;
    width: 100%;
    overflow-y: hidden;
    overflow-x: scroll;
    gap: 0 20px;
    grid-template-rows: auto auto;
    grid-auto-columns: auto;
    grid-auto-flow: column;
    /* standards way */
    scrollbar-color: #b5b5b5 #616161;
    scrollbar-width: thin;
    scroll-snap-type: x mandatory;
    scroll-behavior: smooth;
}
```

Nice! Sadly, this is something that images just don't convey. However, if you open `example_16-03`, then you can have a play with it for yourself.

We've taken a basic scrolling panel and, with some choice CSS, managed to add custom-colored scrollbars, smooth scrolling, and snap points for a very impressive and robust user experience.

We've nearly reached the end of the bonus techniques section. But there was one more thing I stumbled upon in the last year or so that eluded me for years and has proved to be a technique I keep coming back to. It was demonstrated by a Codepen user called Lasse, and you can view his original here: `https://codepen.io/lasjorg/pen/zYaEVWQ`.

Rounded gradient borders with semi-transparent backgrounds

The technique allows you to get a rounded gradient border on an element, while crucially also allowing you to have a semi-transparent or blurred background.

You can view the finished example as `example_16-04`. Heck, just because we can, I made the border animate too, so open it in a browser if you can. But to give you an idea, it looks like this:

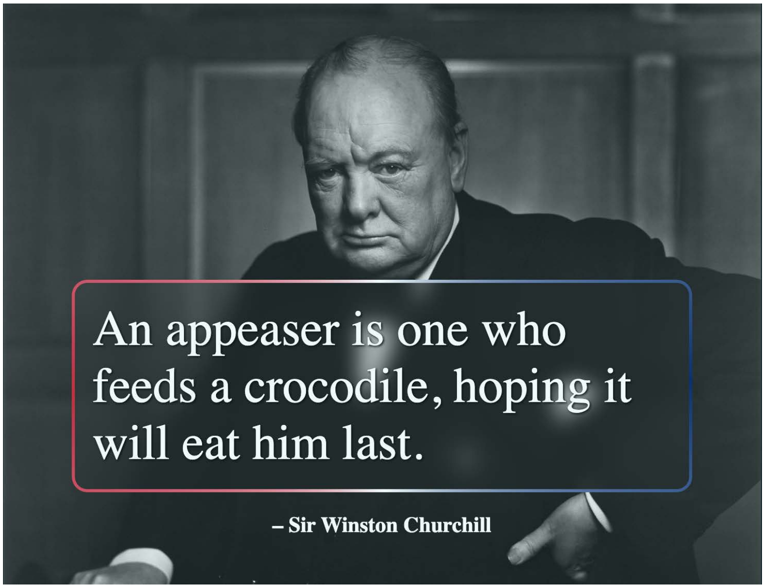

An appeaser is one who feeds a crocodile, hoping it will eat him last.
– Sir Winston Churchill

Figure 16.8: It's possible to have a transparent background and a rotating, rounded border gradient

I'll only detail the code here relevant to the rounded border on the semi-transparent background.

Our markup for the quote looks like this, but you could use this technique on pretty much any markup:

```html
<figure class="author">
    <blockquote class="gradient-border-mask">
        <p>
            An appeaser is one who feeds a crocodile,
            hoping it will eat him last.
        </p>
    </blockquote>
    <figcaption>- Sir Winston Churchill</figcaption>
</figure>
```

And here is the corresponding CSS:

```css
.gradient-border-mask {
    position: relative;
    backdrop-filter: blur(10px);
}

.gradient-border-mask::before {
    content: "";
    position: absolute;
    inset: 0;
    border-radius: 15px;
    border: 3px solid transparent;
    background: conic-gradient(
            from var(--angle),
            #c8102e,
            white,
            #012169,
            white,
            #c8102e
        )
        border-box;
    animation: rotateAngle 5s forwards linear infinite;
    mask: linear-gradient(#fff 0 0) padding-box,
        linear-gradient(#fff 0 0);
    mask-composite: exclude;
}
```

I have kept the background of the area simple, just applying a blur, but you might choose a semi-transparent color or gradient instead, or both!

The clever stuff happens with the `::before` pseudo-element. The background of the `::before` pseudo-element is what creates the border you will ultimately see. Note that the background is set to `border-box` after the gradient, so it extends out behind the border area of the element (3px in this example). You could also position the background behind the border area by using `background-size` and a `calc()` equal to the width of the border, such as `calc(100% + 3px)`, but I find using `border-box` on the `background` far more elegant. As this is just a rectangle currently, you can add `border-radius`, except it won't be a border without a little extra secret sauce.

Here is the clever bit I had never seen before; we then use `mask` and `mask-composite`. This allows us to create a *shape* that our gradient border will inhabit. To create this shape, we need to composite two images together and find the difference, excluding one from the other. That might sound like a lot of work, but we can make those two images with CSS using a gradient. It doesn't matter that the `linear-gradient` used here is actually just a flat white color; the fact that it is defined as `linear-gradient` means that the browser renders the outcome of that notation as an image, and the image can be composited. So, the first mask is a `linear-gradient` set to `padding-box`, which then crucially does not include the border, and the second gradient is the full size, and the difference between them, once one is excluded from the other, appears as the border shape. Genius!

Hopefully, with the extras we have covered in this chapter, you have a few more handy tricks in your arsenal for the next project you embark on.

Parting advice

All that is left in the remainder of the chapter, and indeed the book (please don't cry—you'll start me off), is for me to offer what I feel are the most important considerations for any responsive web project. Here goes.

Getting designs in the browser as soon as possible

The more responsive web work I have done, the more important I have found it to get designs up and running in a browser environment as soon as possible. If you are a designer as well as a developer, then that simplifies matters. As soon as you have enough of a feel visually for what you need, you can get it prototyped and develop the idea further in a browser environment.

If you are primarily a developer, then this can aid the design process hugely in order to get the design living and breathing in the browser. Without fail, every project that I work on gets revised in some way as a result of the designs being built in the browser. That isn't a failure of the original flat designs; it's a consequence of realizing that the design is just a step toward realizing an idea. And seeing it working in a browser is one step closer.

There is a slew of problems that can only be solved by seeing the design in a browser. What about all those device and browser window sizes? What about the interactivity that occurs when the user opens a menu? How long should that introductory animation run for? And is it too much anyway? These are the kind of design decisions that can only be made when the design is realized in a browser.

Testing on real devices

If you can, start to build up a "device lab" of older devices (phones/tablets) to view your work on. Having a number of varied devices running different operating systems and versions is hugely beneficial. Not only does it let you feel how a design actually works across different devices, but it also exposes layout/rendering peculiarities earlier in the process. After all, no one enjoys believing that they have finished a project only to be told it doesn't work properly in a certain environment. Test early, and test often! Buying extra devices need not cost the earth. For example, you can pick up older phone and tablet models on eBay, or buy them from friends/relatives as they upgrade.

Embracing progressive enhancement

In previous chapters, we have considered the notion of progressive enhancement. It's an approach to development that I have found so useful in practice that I think it bears repeating. The fundamental idea of progressive enhancement is that you begin all your frontend code (HTML, CSS, or JavaScript) with the lowest common denominator in mind. Then, you progressively enhance the code for more capable devices and browsers. That may seem simplistic, and it is, but if you are used to working the other way around, then by designing the optimum experience and then figuring out a way of making that thing work on lesser devices/browsers, you'll find progressive enhancement an easier approach.

Imagine a lower-powered, poorly featured device, stuck on some older browser version where the newer CSS and web platform features are unavailable. In that scenario, what can you do to provide a usable experience? Most importantly, you should write meaningful HTML5 markup that accurately describes the content. This is an easier task if you're building text- and content-based websites. In that instance, concentrate on using elements such as `main`, `header`, `footer`, `article`, `section`, and `aside` correctly. Not only will it help you discern different sections of your code, but it will also provide greater accessibility for your users at no extra cost.

If you're building something like a web-based application or visual UI components (carousels, tabs, accordions, and the like), you'll need to think about how to distill the visual pattern down into accessible markup.

The reason good markup is so crucial is that it provides a base-level experience for all users. The more you can achieve with HTML, the less you have to do in CSS and JavaScript to support older browsers. And nobody, and I really mean nobody, likes writing the code to support older browsers.

It's by no means a simple feat to think in this manner. However, it is an approach that is likely to serve you well in your quest to do as little as possible to support ailing browsers. Now, about those browsers…

Defining a browser support matrix

Knowing the browsers and devices a web project needs to support upfront can be crucial to developing a successful responsive web design. We've already considered why progressive enhancement is so useful in this respect. If done correctly, it means that the vast majority of your site will be functional on even the oldest browsers.

However, there may also be times when you need to start your experience with a higher set of prerequisites. Perhaps you are working on a project where JavaScript is essential, which is not an uncommon scenario. In that instance, you can still progressively enhance, but you are merely enhancing from a different starting point.

Whatever your scenario, the key thing is to establish what your starting point is. Then and only then can you define and agree on what visual and functional experiences the different browsers and devices that you intend to support will get.

Functional parity, not visual parity

It's arguably unrealistic to get any website looking and working the same in every browser. Besides quirks that are specific to certain browsers, there are essential functional considerations. For example, we have to consider things such as touch targets for buttons and links on touchscreens that aren't relevant to mouse-based environments.

Therefore, some part of your role as a responsive web developer is to educate whoever you are answerable to (your boss, client, or shareholders) that "supporting older browsers" does not mean "looking the same in older browsers." The line I tend to run with is that all browsers in the support matrix will get functional parity, not visual parity. This means that if you have a checkout to build, all users will be able to go through the checkout and purchase goods. There may be visual and interaction flourishes afforded to the users of more modern browsers, but the core task will be achievable by all.

Plus, we are in somewhat of a golden era when it comes to support for CSS features. There are still some features you can only get in one browser, but the overwhelming majority of important features nowadays enjoy broad support.

> Browser vendors themselves have even had programs to ensure greater interoperability. Read more about that here: `https://web.dev/blog/interop-2025/`.

Choosing the browsers to support

Typically, when we talk about which browsers to support, we're talking about how far back we need to look. Here are a couple of possibilities to consider, depending on the situation.

If it's an existing website, take a look at the visitor statistics (for example, Google Analytics or similar). Armed with some figures, you can do some rough calculations. For example, if the cost of supporting browser X is less than the value produced by supporting browser X, then support browser X!

Also, if there are browsers in the statistics that represent less than 10% of users, look further back and consider any trends. How has usage changed over the last 3, 6, and 12 months? If it's currently 6% and that value has halved over the last 12 months, then you have a more compelling argument to consider ruling that browser out for specific enhancements.

If it's a new project and statistics are unavailable, I usually opt for a "previous two" policy. This would be the current version, plus the previous two versions of each browser. For example, if Safari 28 was the current version, look to offer your enhancements for that version, plus Safari 26 and Safari 27 (the previous two). This choice is easier with "evergreen" browsers, which is the term given to browsers that continually update on a rapid release cycle (Firefox, Chrome, and Microsoft Edge, for example).

Tiering the user experience

Let's assume that your shareholders are educated and on board. Let's also assume that you have a clear set of browsers that you would like, and need, to add enhanced experiences for. We can now set about tiering the experience. I like to keep things simple, so, where possible, I opt to define a simple base tier and a more enhanced tier.

Here, the base experience is the minimum viable version of the site, and the enhanced version is the most fully featured and aesthetically pleasing version. You might need to accommodate more granularity in your tiers, but regardless of how the tiers are defined, ensure that you define them and what you expect to deliver with each. Then, you can go about coding those tiers. That's where techniques such as feature queries, which we covered in *Chapter 6*, *CSS Nesting, Layers, Selectors, and More*, will come in handy.

Avoiding CSS frameworks in production

There are a plethora of free CSS frameworks available that aim to aid in the rapid prototyping and building of responsive websites. The most common example is Bootstrap (`https://getbootstrap.com`). While Bootstrap is a well-tested project, particularly for learning how to build responsive visual patterns, I think any CSS framework should be avoided in production.

I've spoken to plenty of developers who start all of their projects with Bootstrap or some other visual frameworks and then amend them to fit their needs. This approach can be incredibly advantageous for rapid prototyping (for example, to illustrate some interaction with clients), but I think it's the wrong thing to do for projects you intend to take through to production.

Firstly, from a technical perspective, it's likely that starting with a framework will result in your project having more code than it actually needs. Secondly, from an aesthetic perspective, due to the popularity of these frameworks, it's likely your project will end up looking very similar to countless others.

Finally, if you only copy and paste code into your project and tweak it to your needs, you're unlikely to fully appreciate what's going on "under the hood." It's only by defining and solving the problems you have that you can master the code you place into your projects.

Hiding, showing, and loading content across viewports

One of the commonly touted maxims regarding responsive web design is that if you don't have something on the screen in smaller viewports, you shouldn't have it there in larger ones either.

This means users should be able to accomplish all the same goals (such as buying a product, reading an article, or accomplishing an interface task) at every viewport size. This is common sense. After all, as users ourselves, we've all felt the frustration of going to a website to accomplish a goal and being unable to because we're using a smaller screen.

This also means that as screen real estate is more plentiful, we shouldn't feel compelled to add extra things just to fill the space (widgets, adverts, or links, for example). If the user could live without those extras on smaller screen sizes, they'll manage just fine on bigger ones.

In broad terms, I think the preceding maxim is sound advice. If nothing else, it makes designers and developers question the content they display onscreen more thoroughly. However, as ever in web design, there will be exceptions.

As far as possible, I resist loading in new markup for different viewports, but, very occasionally, it's a necessity. Some complex user interfaces might simply work better with different markup and designs at wider viewports. In that instance, JavaScript is typically used to replace one area of markup with another. It isn't the ideal scenario, but it is sometimes the most pragmatic.

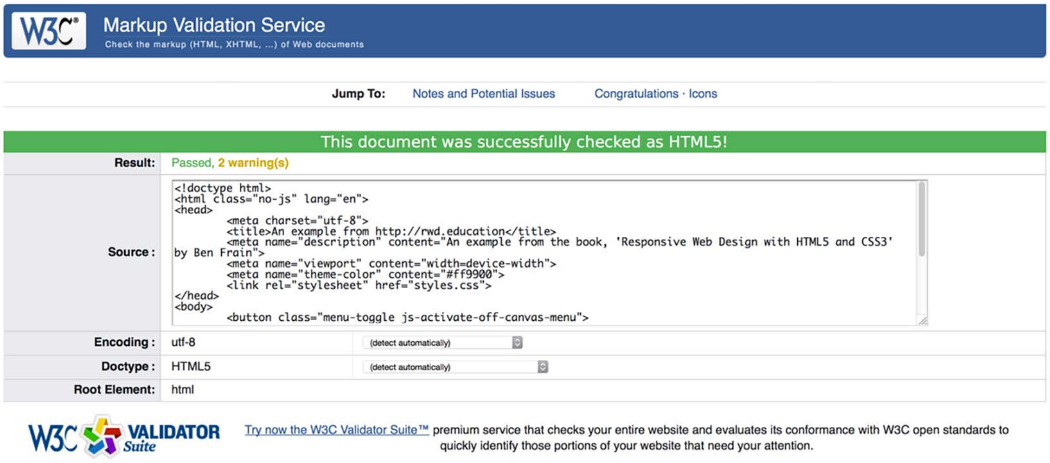
By the same token, I think it's perfectly reasonable to have sections of markup hidden in CSS until they are appropriate. I'm thinking about things such as headers. On more than one occasion, before Grid was well supported, I wasted an inordinate amount of time trying to figure out how to convert one header layout for mobile into a different layout for larger screens, all while using the same markup. It was folly! The pragmatic solution is to have both pieces of markup and just hide each as appropriate with a media query. When the difference is a handful of elements and the associated styles, there are almost always more sensible savings to make elsewhere.

These are the choices you will probably face as you code more and more complex responsive web designs, and you'll need to use your own judgment as to what the best choice is in any given scenario. However, it's not a cardinal sin if you toggle the visibility of the odd bit of markup with `display: none` to achieve your goal.

Validators and linting tools

Generally speaking, writing HTML and CSS is pretty forgiving. You can nest the odd thing incorrectly, miss the occasional quotation mark or self-closing tag, and not always notice a problem. Despite this, on an almost weekly basis, I manage to befuddle myself with incorrect markup. Sometimes, it's a slip-up, such as accidentally typing an errant character. Other times, it's a schoolboy error, such as nesting a `div` inside a `span` (invalid markup, as a `span` is an inline element and a `div` is a block-level element, which leads to unpredictable results).

Thankfully, there are great tools to help out. At worst, if you're encountering a weird issue, head over to `https://validator.w3.org/` and paste your markup in there. It will point out any errors along with line numbers, helping you to easily fix things up:


W3C
Markup Validation Service
Check the markup (HTML, XHTML, ...) of Web documents
Jump To: Notes and Potential Issues Congratulations · Icons
W3C VALIDATOR Suite

This document was successfully checked as HTML5!

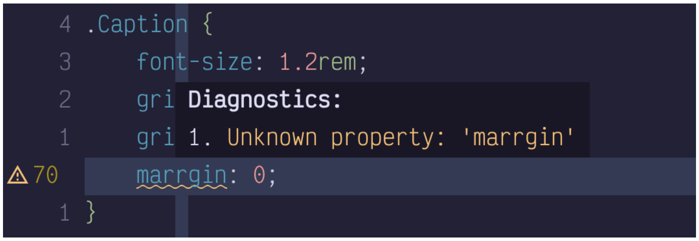
Result:	Passed, 2 warning(s)
Source :	see below
Encoding :	utf-8 (detect automatically)
Doctype :	HTML5 (detect automatically)
Root Element:	html

```
<!doctype html>
<html class="no-js" lang="en">
<head>
        <meta charset="utf-8">
        <title>An example from http://rwd.education</title>
        <meta name="description" content="An example from the book, 'Responsive Web Design with HTML5 and CSS3'
by Ben Frain">
        <meta name="viewport" content="width=device-width">
        <meta name="theme-color" content="#ff9900">
        <link rel="stylesheet" href="styles.css">
</head>
<body>
        <button class="menu-toggle js-activate-off-canvas-menu">
```

Try now the W3C Validator Suite™ premium service that checks your entire website and evaluates its conformance with W3C open standards to quickly identify those portions of your website that need your attention.

Figure 16.9: Skip the validation of HTML at your peril!

Better still, install and configure **linting** tools for your HTML, CSS, and JavaScript. Or, choose a text editor with some sanity checking built in. Then, problem areas are flagged up in your code as you go. Here's an example of a simple spelling error in CSS flagged up by the **Language Server Protocol** (**LSP**) in the Neovim editor:



```
.Caption {
    font-size: 1.2rem;
    gri Diagnostics:
    gri 1. Unknown property: 'marrgin'
⚠   marrgin: 0;
}
```

Figure 16.10: Embrace tooling that saves you from basic mistakes

Like a clown, I've clumsily typed `marrgin` instead of `margin`. The editor has spotted this fact and pointed out the error of my ways. Embrace these tools where possible. There are better uses of your time than tracking down simple syntax errors in your code.

Performance

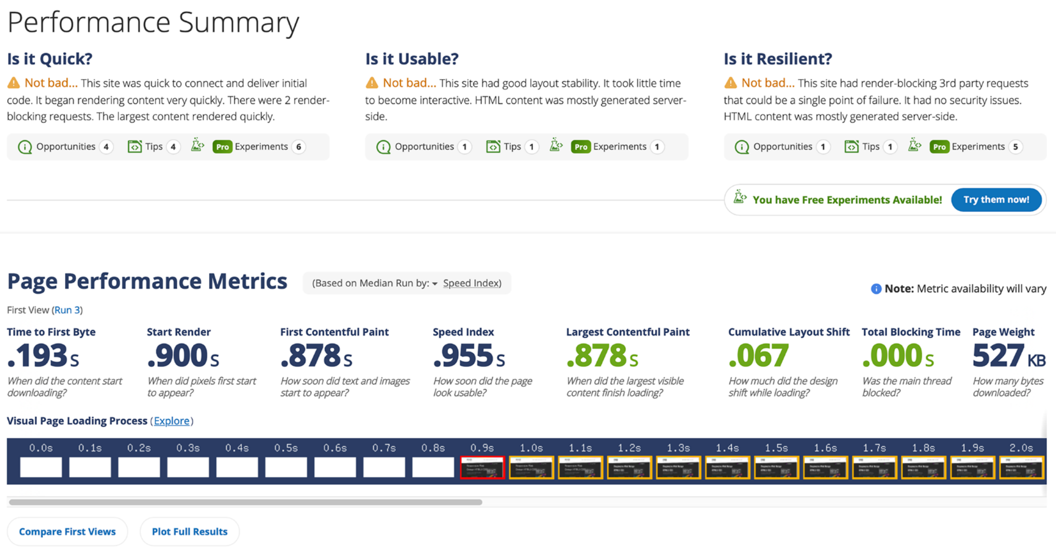
Considering the performance of your responsive web designs is as important as the aesthetics. However, performance presents something of a moving target. For example, browsers update and improve the way they handle assets, new techniques are discovered that supersede existing "best practices," and technologies eventually get enough browser support that they become viable for widespread adoption. The list goes on.

There are, however, some basic implementation details that are pretty solid advice. These are the following:

- Minimize the page weight. If you can compress images to a fraction of their original size, you should. This should always be your first task with optimizing. It's possible to double the file size savings by compressing one image compared to compressing and minifying all of your CSS and JavaScript. We looked at image optimization more fully in *Chapter 10*, *Responsive Images*.
- Defer non-essential assets. If you can load any additional CSS and JavaScript after the page has initially rendered, it can greatly reduce the perceived load time.
- Ensure that the page is usable as soon as possible, which is often a byproduct of doing all the preceding points.

Performance tools

There are great tools available to measure and optimize performance. My personal favorite is `https://webpagetest.org`. At its simplest, you can pick a URL and click on **Start Test**. It will show you a complete analysis of the page, but even more usefully, if you choose the visual comparison option, it shows a "filmstrip" view of the loaded page, allowing you to concentrate on getting the rendered page completed sooner. Here's an example of the "filmstrip" view of the previous version of the `https://rwd.education` home page:



Performance Summary

Is it Quick?

Not bad... This site was quick to connect and deliver initial code. It began rendering content very quickly. There were 2 render-blocking requests. The largest content rendered quickly.

Opportunities 4 Tips 4 Pro Experiments 6

Is it Usable?

Not bad... This site had good layout stability. It took little time to become interactive. HTML content was mostly generated server-side.

Opportunities 1 Tips 1 Pro Experiments 1

Is it Resilient?

Not bad... This site had render-blocking 3rd party requests that could be a single point of failure. It had no security issues. HTML content was mostly generated server-side.

Opportunities 1 Tips 1 Pro Experiments 5

You have Free Experiments Available!

Page Performance Metrics (Based on Median Run by: Speed Index)

Note: Metric availability will vary

First View (Run 3)

Time to First Byte	Start Render	First Contentful Paint	Speed Index	Largest Contentful Paint	Cumulative Layout Shift	Total Blocking Time	Page Weight
.193s	.900s	.878s	.955s	.878s	.067	.000s	527KB
When did the content start downloading?	When did pixels first start to appear?	How soon did text and images start to appear?	How soon did the page look usable?	When did the largest visible content finish loading?	How much did the design shift while loading?	Was the main thread blocked?	How many bytes downloaded?

Visual Page Loading Process (Explore)

Figure 16.11: Seeing the page load as a filmstrip can really tell the story of performance

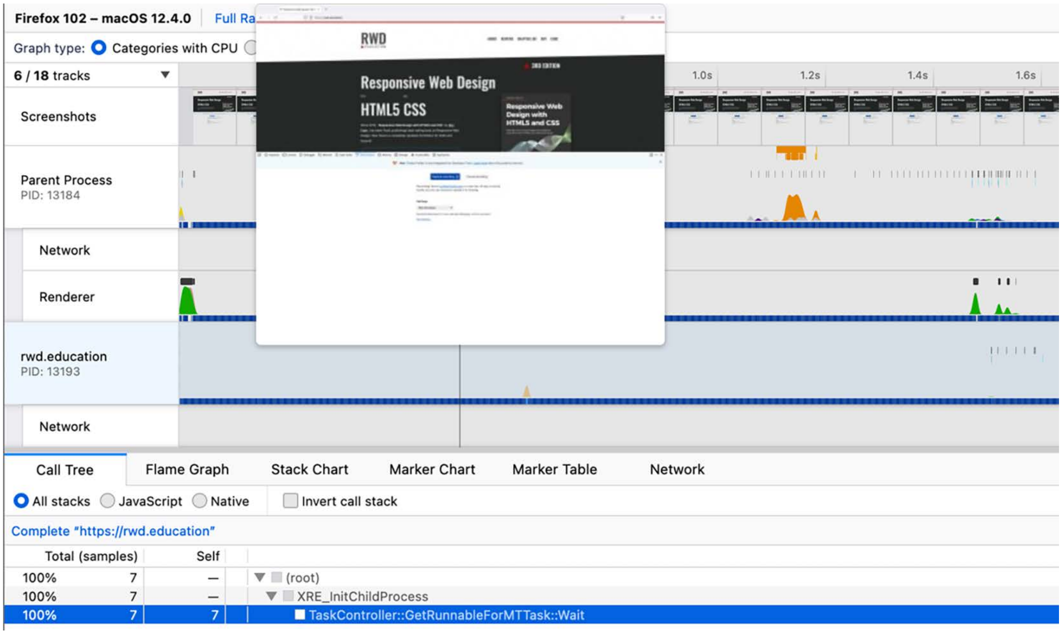
Even the free tier of this service will offer practical advice on what to do to make your website load faster and be more accessible. I learn something new almost every time I test a site there!

There are also increasingly powerful performance tools built into browser developer tools, although you should be aware that these can be a little intimidating at first use. It can feel a little like opening something such as After Effects for the first time and wondering what on earth you are supposed to do!

Delving into the developer tools in any detail is beyond our discussion scope here, but let's consider some of the kinds of information you can get at to improve the performance of your site.

If you're working on something that uses a lot of JavaScript or video, which is sure to negatively affect the device battery, then the CPU timeline feature of Safari (macOS only, unfortunately) may be of interest: `https://webkit.org/blog/8993/cpu-timeline-in-web-inspector/`.

Firefox has the Profiler tool, which mercifully has a full guide here: `https://profiler.firefox.com/docs/#/`. It can give you incredibly detailed information, including the same kind of filmstrip feedback we saw in Web Page Test. I'll confess to finding it a little unapproachable, especially when we are typically looking for the "low-hanging fruit" when it comes to fixing up our sites.


Firefox 102 – macOS 12.4.0
Full Ra
RWD
Responsive Web Design
HTML5 CSS
6 / 18 tracks
1.0s
1.2s
1.4s
1.6s
Screenshots
Parent Process
Network
Renderer
Network
Call Tree
Flame Graph
Stack Chart
Marker Chart
Marker Table
Network

Graph type: ● Categories with CPU ○

Parent Process PID: 13184

rwd.education PID: 13193

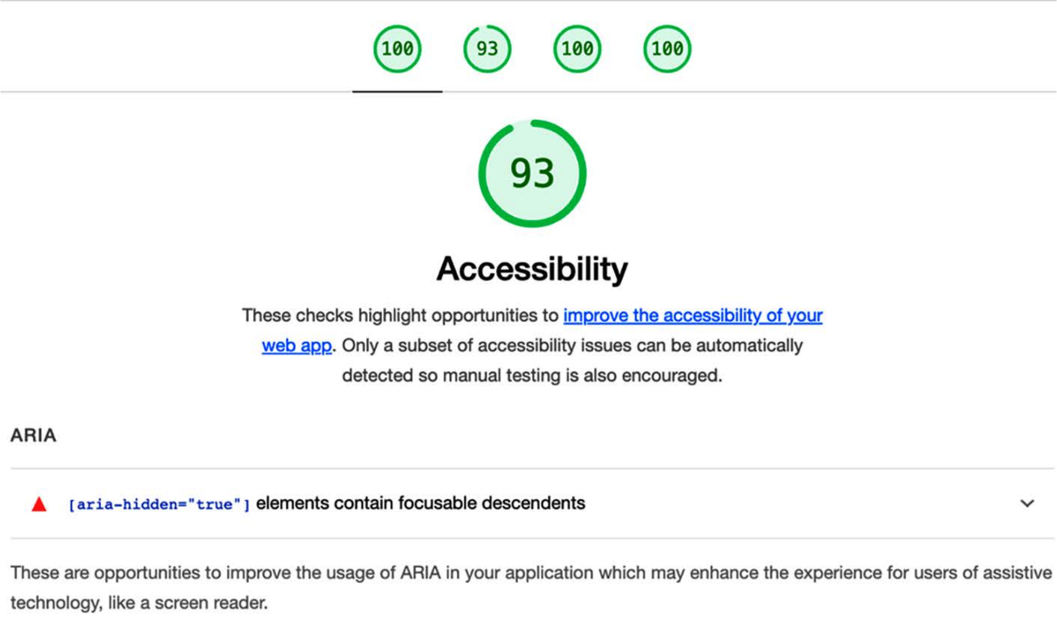
● All stacks ○ JavaScript ○ Native ☐ Invert call stack

Complete "https://rwd.education"

Total (samples)		Self	
100%	7	–	▼ (root)
100%	7	–	▼ XRE_InitChildProcess
100%	7	7	TaskController::GetRunnableForMTTask::Wait

Figure 16.12: Firefox Profiler is powerful but perhaps overkill

Typically, I tend to use Lighthouse, which has its own tab in the Google Chrome Developer Tools. Point it at a site and it will give you a breakdown of where improvements can be made:



Accessibility

These checks highlight opportunities to improve the accessibility of your web app. Only a subset of accessibility issues can be automatically detected so manual testing is also encouraged.

ARIA

▲ `[aria-hidden="true"]` elements contain focusable descendents

These are opportunities to improve the usage of ARIA in your application which may enhance the experience for users of assistive technology, like a screen reader.

Figure 16.13: The Lighthouse tools not only tell you what you can improve, but they tell you how to do it

It's such a great tool, as every piece of advice has links to additional documentation, so you can read more about the problem and potential ways of addressing it.

If you are new to looking at the performance of the sites you are making, my advice would be to start with Lighthouse and go deeper only if you feel you need or want to. If you have scores close to 100 in Lighthouse, you've done a decent job.

> Whenever you try to optimize performance, ensure that you take measurements before you begin (otherwise, you will not understand how effective your performance work has been). Then, make amendments, test, and repeat.

The next big things

One of the interesting things about frontend web development is that things change rapidly. There is always something new to learn, and the web community is always figuring out better, faster, and more effective ways of solving problems.

For example, when writing the first edition of this book, over 10 years ago, responsive images (`srcset` and the `picture` element, which are detailed in *Chapter 10*, *Responsive Images*) simply didn't exist. Back then, we had to use clever third-party workarounds to serve up more appropriate images to different viewport sizes. Now, that common need has been rationalized into a W3C standard, which has been implemented across browsers.

Things such as CSS nesting that were new back in 2010, and used to require a CSS preprocessor such as Sass to enjoy, are baked in. In the last chapter, we looked at newer features still, including anchor positioning and scroll-driven animation—incredible new capabilities that were just a dream a few years back.

Now I'm starting to sound like one of those old guys who looks across a city and says, "*I remember when all this was fields!*" but the point I am trying to make is this…

Progress on the web shows little sign of abating. Even now, things like **WebAssembly**, or **WASM** as it is often referred to, are gaining more and more traction. WebAssembly is a means of having web code that runs far more akin to the speed of a compiled language. In short, it will make things on the web feel a whole lot faster. Lin Clark, a developer at the browser-maker Mozilla, has a very accessible series on WebAssembly here: `https://hacks.mozilla.org/2017/02/a-cartoon-intro-to-webassembly/`. Design tools you may be familiar with, such as Figma, make extensive use of WebAssembly: `https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/`.

And due to the current infancy and few implementations, in this book, we didn't even look at things such as custom functions and mixins for "logic" in CSS, or a bunch of new properties to make carousels work with far less work; you can read all about those in the *CSS Overflow Module Level 5* specification here: `https://drafts.csswg.org/css-overflow-5/`. Maybe these are things we can explore together in the next edition.

But, in short, expect change and new things from the web platform, and embrace it!

Summary

As we reach the end of our time together, your humble author hopes to have related all the essential techniques and tools you'll need to start building your next website or web application responsively.

It's my conviction that, by approaching web projects with a little forethought and by potentially making a few modifications to existing workflows and practices, it's possible to embrace modern techniques that provide fast, flexible, accessible, and maintainable websites that can look incredible, regardless of the device used to visit them.

We've covered a wealth of information in our time together: techniques, technologies, performance optimizations, specifications, workflow, tooling, and more. I wouldn't expect anybody to absorb it all in one read.

Therefore, the next time you need to remember this or that syntax, or to refresh your mind about one of the responsive-related subjects we've covered, I hope you'll dip back into these pages; I'll be right here waiting for you.

Until then, I wish you good fortune in your responsive web design quests.

See you again sometime.

`packt.com`

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

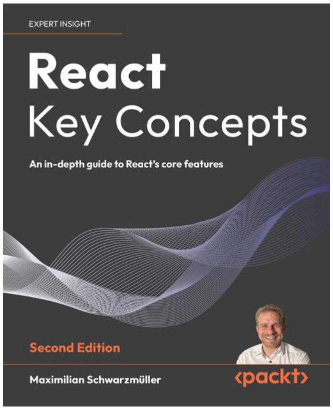
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At `www.packt.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

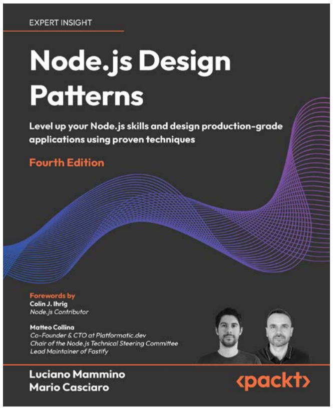

EXPERT INSIGHT
React
Key Concepts
An in-depth guide to React's core features
Second Edition
Maximilian Schwarzmüller
‹packt›

React Key Concepts

Maximilian Schwarzmüller

ISBN: 9781836202271

- Build modern, user-friendly, and reactive web apps
- Create components and utilize props to pass data between them
- Handle events, perform state updates, and manage conditional content
- Add styles dynamically and conditionally for modern user interfaces
- Use advanced state management techniques such as React's Context API
- Utilize React Router to render different pages for different URLs
- Understand key best practices and optimization opportunities
- Learn about React Server Components and Server Actions



Node.js Design Patterns

Luciano Mammino, Mario Casciaro

ISBN: 9781803238944

- Understand Node.js basics and its async event-driven architecture
- Write correct async code using callbacks, promises, and async/await
- Harness Node.js streams to create data-driven processing pipelines
- Implement trusted software design patterns for production-grade applications
- Write testable code and automated tests (unit, integration, E2E)
- Use advanced recipes: caching, batching, async init, offload CPU-bound work
- Build and scale microservices and distributed systems powered by Node.js

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Responsive Web Design with HTML5 and CSS — Fifth Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

<a> element 30, 31
  reference link 30

<b> element 43

<dialog> element 55-57

<em> element 44

<i> element 44, 45

<span> element 43

<strong> element 44

@font-face
  used, for implementing web fonts 227, 228

@font-face CSS rule 226
  reference link 227

:has() relational pseudo-class 212, 213

!important declaration 186, 187
  in cascade layers 188, 189

:is() functional pseudo-class 210, 211

:last-child selector 197

@media (color-gamut) media query 263

:root pseudo-class 404

@scope
  application of selectors, limiting 214
  for targeting elements, to get less specificity 215
  practicality 218
  specificity 216
  using, inside style tag 217

@starting-style
  used, for item transition entering DOM 395, 396

@supports
  using, as proxy 222
  using, to fork CSS 220

:where() functional pseudo-class 211, 212

A

Accessible Rich Internet Applications (ARIA) 49, 50

address elements 42

advanced media query considerations 75
  container queries 78
  file size 78
  nesting media queries 78
  own style sheets 76
  render-blocking asset 75
  style sheets, consolidating 76, 78

advanced switching
  with sizes 325, 326
  with srcset 325, 326

AI
  coding with 8

align-items property 113

align-self property 114, 115

alpha channels 255
  defining, with hex 255
  using, with functional notation 256, 257

alternate media sources
  providing 52, 53

anchor
  positioning 464
  relationship, creating 465-467

anchored content
  positioning, with position-area 467, 468
Anchoreum game
  reference link 464
anchor() functions
  used, for positioning 469, 470
anchor-size() function
  used, for sizing anchored elements 470, 471
anchor tag 30
angle gradients 284
animation-composition property 400
animation-fill-mode property 393, 394
  reference link 394
animation-range property 480, 481
animation-timeline property 480
art direction
  with picture element 326, 327
article element 34
aside element 34
associated datalist element 426, 427
audio and video tags 53
  aspect ratio, maintaining 54
  iframes 53, 54
  responsive HTML5 video 53, 54
autocomplete attribute 425
auto-fill 156-158
auto-fit 156-158
autofocus attribute 424
AVIF image format 319

B

backdrop-filter property 300
background-blend-mode property 292
background fill effect
  creating 452, 453
background gradient 278
  conic gradient 284-287
  extent keywords, for responsive sizing 283
  linear-gradient notation 279
  radial background gradient 281, 282
  repeating gradient 287
background gradient patterns 288, 289
Bennett Feely's Clippy
  reference link 306
blockquote element 37, 38
Bohemian Coding's Sketch
  reference link 336
Bootstrap
  reference link 515
box-shadow property
  reference link 278
box shadows 276
  inset shadow 276
  multiple box shadow 277
  spread value 277
breakpoint 97
browser
  deciding, to pick one
    image over another 326
browser environment
  designs, obtaining 511
browser support matrix
  defining 513
  functional considerations 513
  selecting 514
  user experience 514

C

calc() function 412, 413
calc-size() function
  used, for animating height 399

Candidate Recommendation (CR) 79

Can I Use
  reference link 6

caret 429

caret-color CSS property
  used, for styling input caret 429

caret symbol 193

cascade layers 182-184
  declaring 184, 185
  !important declaration 188, 189
  naming 184
  nesting, inside other layers 186
  old browser support 189
  priority of rules, in and outside layers 185
  reference link 188
  rules, adding 185

Character Data marker (CDATA) 352

child combinator selector 206

clamp() function 415
  reference link 416

clipping source 306

CMYK color space 258

CodePen
  reference link 347

color
  concepts and terminology 257-260

colored fonts 248

color formats 261

color-mix() function 267, 268
  reference link 268

color space 258

color stops 280, 281

color type input 440

combinator selectors 205
  child combinator selector 206
  empty (:empty) selector 208, 209
  negation (:not) selector 207, 208
  next sibling selector 206
  subsequent sibling selector 207

comps 98

conditionals
  combining 222, 223

conic gradient 284-287

container queries 84, 85
  container relative lengths 91, 92
  named container 90, 91
  using 85-90

container relative lengths 91, 92

container style queries 92
  with CSS custom properties 94, 95
  working, on computed style property 93

content, across viewports
  displaying 515, 516
  hiding 515, 516
  loading 515, 516

context selector 178

contrast-color() function 268
  original 268
  updates version 269
  version flaws 269

control flow 66

CPU timeline feature
  reference link 519

cross-document view transitions 491-493

CSS
  SVG properties 353
  SVG values 353
  Unicode characters, inserting 45, 46
  used, for animating SVG 353, 355
  used, for styling HTML5 forms 446-448

CSS 2D transforms 373, 374
  matrix 380, 381
  module 373
  rotate 378, 379
  scale 374
  skew 379, 380
  translate 375
  translate, using to center absolutely positioned elements 376-378
CSS 3D transformations 383-386
  translate3d example 387-390
  translate3d property 386, 387
CSS3 Media Queries module
  CSS conditional logic 66
  reference link 66
CSS anchor positioning
  reference link 475
CSS animation 365, 390-393
  animation-fill-mode property 393, 394
CSS attribute selectors 192, 193
  reference link 196
  substring matching attribute selector 193
CSS background patterns
  reference link 288
CSS basic shapes 302
  clip-path with circle 303
  clip-path with ellipse 304
  clip-path with inset 304, 305
  clip-path with polygon 305, 306
  clip-path with URL 306, 307
  reference link 302
CSS capabilities 190
CSS clip-path 302
  animating 310, 311
  CSS shape() function 307-309
CSS color
  current state 260, 261
  display-p3 color space 261, 262
  @media (color-gamut) media query 263
  P3 262
CSS conditional logic 66
CSS Conditional Rules Module Level 3
  reference link 220
CSS custom properties
  SVGs, recoloring with 345-347
CSS filters 294, 295
  combining 300
  selecting 295-299
CSS frameworks
  avoiding 515
CSS functions 411, 412
  calc() function 412, 413
  clamp() function 415
  max() function 414, 415
  min() function 414
CSS Gradient Generator
  reference link 284
CSS Grid 134
CSS Images Module Level 3 specification
  reference link 279
CSS Media Queries Level 5 79
  features 72
  future 83
  hover media feature 80
  interaction media feature 79
  pointer media feature 80
  prefers-color-scheme media feature 81
  prefers-reduced-motion feature 82
  range syntax 83
  user preference media feature 81

CSS nesting 176
- adjacent sibling and data-attribute example 179, 180
- ancestor relationship, writing 179
- media queries, nesting 180, 181
- nested selectors example 180
- & symbol 178
- working 177

CSS Object Model (CSSOM) 230

CSS performance 300-302
- reference link 301

CSS rule
- anatomy 190

CSS Scroll Snap 505
- reference link 505
- scroll-snap-align property 506
- scroll-snap-stop property 506
- scroll-snap-type property 505, 506

CSS selectors 190

CSS shape() function 307-309

CSS structural pseudo-classes 196
- :last-child selector 197
- of selector 202, 203
- nth-based selection, in responsive web designs 203, 204
- nth-child selector 197

CSS substring matching attribute selectors 193
- beginning with 193
- chaining 195, 196
- contains an instance of 194
- ends with 194

CSS transform 365

CSS transition 365-368
- multiple properties, declaring 370
- properties 368, 369
- shorthand property 369

CSS units 190

CSS values
- reference link 413

custom axis 244

custom ident 153

custom properties
- env() environment variables 410
- fallback value, setting up 409
- specificity 408, 409
- theming 405, 406
- values, obtaining with JavaScript 407, 408
- values, setting with JavaScript 407, 408
- web components 410, 411
- working with 404, 405

D

datalist specification
- reference link 428

data- type attribute 192
- reference link 192

Data Uniform Resource Identifier (URI) 339, 340

date and time input type
- date input type 441, 442
- month input type 443
- range input type 445, 446
- time input type 444
- week input type 443

date and time input type 441

date input type 441, 442

dense keyword 150

details element 39-42

details elements
- accordion effect, creating with multiple linked 42

device lab
  testing 512
device pixel ratio (DPR) 326
display-p3 color space 261, 262
div element 37
doctype 28
dots per inch (DPI) 326
DRAW SVG
  reference link 337

E

email input type 430-432
Embedded content specification
  reference link 323
Embedded OpenType 227
embedding media
  in HTML5 50, 52
empty (:empty) selector 208, 209
env() environment variables 410
  reference link 410
environment variables 410
escaping 155
Extensible Markup Language (XML) 332
external style sheet
  used, for styling SVG 351, 352

F

fallback
  providing, for older browsers 266, 267
feature queries 220
  used, for forking code in CSS 220, 221
figcaption element 38
Figma
  reference link 521
figure element 38
Filter Effects Module Level 1
  reference link 294
filters
  SVG, using as 357-360
fixed pixel design
  converting, to fluid proportional layout 98-103
Flash of Invisible Text (FOIT) 230
Flash of Unstyled Text (FOUT) 231
flexbox 103, 104
  alignment properties 111
  bumpy path 104
  characteristics 104
  column reverse 110
  items, laying out vertically 109
  need for 103
  offset items 106, 108
  order of items, reversing 108, 109
  text, centering 104, 105
  wrapping up 131
Flexbox alignment
  cross axis 111
  main axis 111
flexbox alignment properties 111-113
  align-items property 113, 114
  align-self 114
  flex property 118-120
  footer 120, 122
  justify-content property 115-117
  possible alignment values 115
flex property 118-120
flex-wrap 127-130
Fontastic.me
  reference link 337

font-display 230, 231
  reference link 231
  used, for optimizing font load 229, 230
  with font-size-adjust property 232
  with size-adjust property 232
font-display, values
  auto 231
  block 231
  fallback 231
  optional 231
  swap 231
font load
  optimizing, with font-display 229, 230
font optimization technique
  reference link 230
font-size-adjust property 232, 234
font stacks 226
font-variant-numeric
  options 246
footer element 35
frontend web development 520
fr unit 148
functional notation
  alpha channels, defining with 256, 257

G

gamut 258, 260
gap property 130
Glyphhanger
  reference link 228
gradient direction
  specifying 279
graphical objects reusage, from symbols 341-343
  colors, using in different contexts 344
  dual-tone icons, creating 344
  from external sources 347, 348
  SVGs, recoloring with CSS custom properties 345-347
Graphical User Interface (GUI) 336
GreenSock
  reference link 355
  used, for animating SVG 355-357
GreenSock Animation Platform
  reference link 355
Grid
  area 135
  explicit 135-141
  implicit 135-141
  nested elements, allowing 164-166
  syntax 134
  track 135
grid-auto-columns property 141-143
grid-auto-flow property 143, 144
grid-auto-rows property 141-143
grid-gap property 147
grid items
  placing 148
Grid Layout
  items, placing 144-146
  items, sizing 144-146
  setting up 136-140
grid shorthand 160, 161
grid shorthand value 161-163
grid-template-areas 154-156
grid-template-columns 158
grid-template shorthand 160
grouping selectors 209, 210
  :has() relational pseudo-class 212, 213
  :is() functional pseudo-class 210, 211
  :where() functional pseudo-class 211, 212

H

h1-h6 elements 35, 36

header element 35

headings element 36

height

- animating, with calc-size() function 399, 400

hex

- alpha channels, defining with 255

hgroup element 36

high-resolution background images 293

horizontal scrolling panel

- creating 499

horizontal scrolling panels

- creating 498-501
- CSS scroll-behavior 507, 508
- proprietary -webkit- scrollbar styling 504
- rounded gradient borders, with semi-transparent backgrounds 508-511
- scrollbar color accessibility 503
- scrollbars, styling 501, 502

hover media feature 80

HTML5 25

- embedding media 50, 52

HTML5 forms

- associated datalist element 426, 427
- autocomplete attribute 425
- autofocus attribute 424
- component parts 421, 422
- input caret, styling with caret-color CSS property 429
- list attribute 426, 427
- placeholder attribute 422
- required attribute 423, 424
- styling, with CSS 447, 448

HTML5 forms 420, 421

HTML5 forms, with CSS

- background fill effect, creating 452, 453
- customizable select styling 455, 456
- default select styling 453-455
- required fields, indicating 448-452
- styling 446

HTML5 input types 430

- color type input 441
- date and time input type 441
- email input type 430, 431, 432
- number input type 432-434
- pattern input attribute 428, 429
- search input type 440
- tel input type 438, 439
- url input type 435-437

HTML5 markup

- forgiving nature 29

HTML markup

- approach 29, 30

HTML pages 27

- character encoding 28
- Doctype declaration 28
- HTML5 markup 29
- sensible approach 29, 30
- tag and lang attribute 28

HTML text-level semantics 43

- <b> element 43
- <em> element 44
- <i> element 44, 45
- <span> element 43
- <strong> element 44

hue, saturation, and brightness (HSB) value 253

hue, saturation, and lightness (HSL) 253-255

- alpha channels 255
- lightness 254
- saturation 254

Hue, Whiteness, Blackness (HWB) 261

HyperText Markup Language (HTML) 25
  conforming features 45
  elements, usage 46-48
  features 45
  non-conforming features 45
  Unicode characters, inserting 45, 46

I

IcoMoon
  reference link 337

iconizr
  reference link 340, 361

iframes 53, 54

image editing packages
  used, for creating SVG 336, 337

image formats
  comparisons 320, 321

image optimization 320

images
  enhancing, with picture elements 321, 323
  reference link 317

image sprites
  generating 340

Inkscape
  reference link 337

Inkscape gallery
  reference link 337

inline-flex 110, 111

input caret
  styling, with caret-color CSS property 429

inset shadow 276

interaction media feature 79

internal styles
  used, for styling SVG 352

J

Jake Archibald
  reference link 356

JavaScript
  custom properties,
    values setting with 407, 408
  custom property,
    values obtaining with 407, 408
  used, for animating SVG 355

justify-content property 115-117

K

keyframe 390

keyframe selector 391

L

lch() function
  LCH, using with 263-265

Lightness, Chroma, Hue (LCH)
  using, with lch() function 263-265

linear background gradients
  reference link 281

linear-gradient notation 279
  color stops 280, 281
  gradient direction, specifying 279

linting tools 516, 517

list attribute 426, 427

loading attribute 54

Logical Properties and Values 213
  reference link 214

lossy optimization 320

M

main element 32, 33

mask-image 311-313

matches any selector 210

matrix 380, 381

max() function 414, 415

max number ranges
  using 434, 435

media queries
  using, for responsive web design 66
  using, to alter design 72-74

media queries, SVG 360, 361

media query syntax 67, 68
  in CSS file 69
  in link tags 68
  list 70
  logic, inverting 69
  multiple expressions, combining 69
  on @import at-rule 68
  target styles 70
  testing 71

meta viewport tag 64, 65

min() function 414

minmax() function 158, 159

min number ranges
  using 434, 435

mix-blend-mode 314
  reference link 314

modern image formats 318

month input type 443

multi-line truncation 497, 498

multiple background images 290
  background position 291, 292
  background shorthand 292
  background size 290
  high-resolution background images 293
  reference link 293

multiple box shadow 277

multiple text shadow 275

N

named container
  need for 90, 91

named grid lines 151-154

nav element 33, 34

negation (:not) selector 207, 208

nesting container style queries 93, 94

new industry benchmarks, stacking up for mobile page speed
  reference link 317

next sibling selector 206

NonVisual Desktop Access (NVDA) 50

nth-based selection
  in responsive web designs 203-205

nth-child selector 197

nth rules 198
  breaking down 198-201

number input type 434
  max number ranges, using 434, 435
  min number ranges, using 434, 435
  step increments, modifying 435

number input type 433

O

oblique keyword 241

OKLab 266
  reference link 266

OKLch 266
  reference link 266

P

pattern input attribute 428, 429

p element 37

picture element
  image formats and sizes, providing 327, 328
  used, for art direction 326, 327
  using, to progressively enhance images 321, 323

Pixels (px) 70
  reference link 325

placeholder attribute 422

placeholder text
  reference link 423
  styling 422, 423

pointer media feature 80

polar coordinates 264

polygon feature design
  reference link 356

Popover API 460
  styling 461-463

position-area property
  used, for positioning anchored content 467, 468

position-try
  flip tactics 473
  used, for creating custom positions 474, 475
  used, for overflow management 471
  using 472, 473

prefers-color-scheme media feature 81

Profiler tool
  reference link 519

progressive enhancement
  embracing 512

Proposed Recommendation (PR) 79

pseudo-classes 191

pseudo-elements 190, 191

R

radial background gradient 281, 282

radial background gradients
  reference link 283

range input type 445, 446

range syntax 83

Recommendation (REC) 79

red, green, and blue (RGB) color 252

registered axis 243, 244

regular expressions (regexes) 428
  reference link 428

relative colors
  with relative color syntax 270

relative color syntax 270
  light-dark() function 271
  number values 270

render-blocking asset 75

repeat() function 147

repeating gradient 287
  reference link 288

required attribute 423, 424

required fields
  indicating 448-451

responsive example 9
  amending, for larger screen 19-22
  basic HTML file 9-12
  breakpoints 17, 18
  images, taming 14-16
  media queries, entering 17
  shortcomings 23
  width/max-width, of images 16, 17

responsive HTML5 video 53, 54

responsive images
  inherent problem 323, 324

responsive web design
  browser and device landscape 4
  browser support 5
  defining 5
  in nutshell 5
  nth-based selection 203-205
  performance 517, 518
  performance tools 518-520
  text editors 7
  tools, for software development 7

root SVG element 334

rotate 378, 379

S

same-document view transitions 484, 485
  default durations, changing 486
  pitfalls 489-491
  selecting, with view-select-class 488
  view-transition-name value, requiring 486, 487

Sass
  reference link 177

Scalable Vector Graphics (SVG) 227-331
  animating, with CSS 353, 354
  animating, with GreenSock 355-360
  animating, with JavaScript 355
  capabilities and oddities 350
  creating, with image editing packages 336, 337
  defs tag 335
  desc tag 335
  g element 335
  graphical objects reusage, from symbols 341-343
  graphics 332, 333
  history 331
  image sprites, generating 340
  img tag, using 338
  implementation tips 362
  inline, inserting 341
  inserting, as background image 339
  inserting, into web pages 338
  media queries 360, 361
  namespace 335
  object tag, using 338, 339
  optimizing 361
  paths 336
  properties, within CSS 353
  recoloring, with mask 348
  reference link 332
  root SVG element 334
  shapes 336
  styling, with external style sheet 351, 352
  styling, with internal styles 352
  time, saving with icon services 337, 338
  title tag 335
  using, as filters 357
  values, within CSS 353

scale 375

scroll-driven animations 478-480
  animation-range property 481
  animation-timeline property 480
  using, with scroll() function 482, 483

scroll-driven animations specification
  reference link 484

scroll() function
  using, in animation 482, 483

scroll-snap-align property
  options 506

scroll-snap-stop property 506

scroll-snap-type property 505, 506

search input type 440

section element 33

self-closing 27

semantic elements, in HTML5 31, 32
  address element 42
  article element 34
  aside element 34
  blockquote element 37, 38
  details element 39-42
  div element 37
  figcaption element 38
  figure element 38
  footer element 35
  h1-h6 elements 35, 36
  header element 35
  headings element 36, 37
  hgroup element 36, 37
  main element 32, 33
  nav element 33, 34
  p element 37
  section element 33
  sub-headings element 36, 37
  summary element 39-42

short block period 230

shorthand syntax 159
  grid shorthand 160, 161
  grid shorthand value, options 161-163
  grid-template shorthand 160

short swap period 230

single-line truncation 496, 497

size-adjust property 234

sizes
  using, for advanced switching 325, 326

skew 379, 380

SMIL animations 350, 351
  reference link 350, 351

soft breaks 240

source order
  changing 122-127

space-separated functional color notation 257

space-separated syntax 257

spanning
  example 149

Speed Matters, for Google Web Search
  reference link 317

spread value 277

Squoosh
  reference link 319

srcset
  using, for advanced switching 325, 326
  using, for simple resolution switching 324, 325

sRGB 258

structural pseudo-classes
  reference link 208

style() function 92

subgrid value 166-170

sub-headings element 36

subsequent sibling selector 207

summary element 39-42

SVG coordinate system
  reference link 334

svg-edit
  reference link 337

SVG icon services 337, 338

SVG insertion method 349
  capabilities 349
  caveats consideration 349

SVG-specific properties
  reference link 353

system fonts 226

T

tel input type 438, 439

text alignment
  with text-box 235, 236

text-box-edge
  values 237

text-box-trim
  values 237

text flow
  controlling, with text-wrap property 238-240

text-overflow property
  reference link 497

text-shadow property
  reference link 275

text shadows 274, 275
  multiple text shadow 275

text truncation 496
  multi-line truncation 497, 498
  single-line truncation 496, 497

text-wrap property 238
  used, for controlling text flow 238-240
  values 239

time input type 444

timing functions 370, 372

transform-origin property 381, 382

transition-behavior
  using 397-399

translate 375
  using, to center absolutely positioned elements 376

translate3d property 386, 387
  using, for progressive enhancement example 387-390

tweening 351

U

UI element state pseudo-classes
  reference link 208

url input type 435-437

user preference media feature 81

V

validator tools 516, 517

variable font 240
  custom axis 244
  features 244, 246
  font-face, modifying 240, 242
  need for 243
  numbering, with font-variant-numeric 246, 247
  registered axis 243, 244

variation axis 243

viewBox
  reference link 334

viewport 334

viewport relative units 218, 219
  dynamic viewport 219
  largest viewport 219
  percentage length 218
  smallest viewport 219

view transitions 484
  cross-document view transitions 491
  same-document view transitions 484

void elements 27

W

W3C validator
  reference link 30

Web Accessibility Initiative - Accessible Rich Internet Applications (WAI-ARIA) 49

WebAssembly
  reference link 521
web components 410
  reference link 411
Web Content Accessibility Guidelines (WCAG)
  accessibility conformance 48
web fonts
  implementing, with @font-face 227, 228
Web Open Font Format (WOFF) 227
web pages
  SVG, inserting into 338
web page test
  reference link 518
WebP image format 319
web typography 225
week input type 443
Working Draft (WD) 79

X

XML namespace (xmlns) 335